Digitized by the Internet Archive
in 2013

http://archive.org/details/gimlreferenceman857henn

Il6n
Math
8
UIUCDCS-R-77-857
no.857
Cop 2

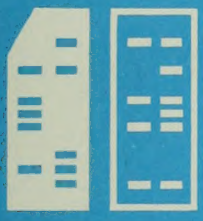
UILU-ENG 77 1709
COO-2383-0038

GIML REFERENCE MANUAL

by

NEAL M. HENNEGAN

April 1977



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the
MAY 13 1977
University of Illinois
at Urbana-Champaign

UIUCDCS-R-77-857

GIML REFERENCE MANUAL*

by

Neal M. Hennegan

April 1977

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

*Supported in part by the Energy Research and Development Administration under contract US ERDA/EY-76-S-02-2383.

TABLE OF CONTENTS

1. Introduction 1

2. Stack Instructions 3

3. Control Transfer Instructions. 15

4. Text Buffer Instructions 25

5. File Management Instructions 43

6. Display Instructions 73

7. Procedure Invocation and Return Instructions 78

8. Screen Coordinate Related Instructions 87

9. IO Related Instructions. 94

10. Debugging Instructions 101

11. Assembler Directives 107

Appendices

A. Instruction Opcode and Assembler Format Summary. 116

1. Introduction

This manual describes in detail the instruction set and assembler directives of the graphical interactive machine language GIML. The instructions are grouped according to dependence on architectural features. Included in the discussion of each instruction are the mnemonic, octal code, description, examples, and notes on assembler processing. For further discussion of the machine architecture and language see GIMLI and GIML--A Machine and Programming Language for Low Cost Interactive Graphical Systems by N. Hennegan.

GAS is an assembler for the GIML language which executes under the UNIX time-sharing system. The table driven assembler features comprehensive diagnostics, efficient one pass scanning, automatic subroutine parameter linkage, stack level monitoring, automatic generation of loads for constants, and an optional program listing. The listing is comprised of the source and three additional columns on the left. The first column is the machine address in octal of the assembled instruction. The second column contains the value of the stack pointer after execution of the instruction. The third column contains the assembled machine code in octal. For readability, byte pairs which are thought of as sixteen-bit values are listed as one word. Due to the one-pass nature of the assembler, forward address references are not known during the generation of the listing and therefore appear as zero. The GIML source is format free and all characters following a "/*" up to the next new line character are considered to be comment.

The GAS assembler recognizes both octal and decimal integer values. Numerical strings having a leading zero are interpreted as octal constants. Octal strings which contain the characters '8' or '9' are illegal. A leading unary minus indicating negation is accepted with both string types.

Identifiers are character strings comprised of the following character set:

{a-z, A-Z, 0-9, $, _}

Identifiers, however, cannot begin with a numeric character. No maximum length is specified although only the first nine characters are used to differentiate between strings.

Text strings are enclosed in double quotes. All ascii characters are allowable except the new-line character, and at present no mechanism is available for imbedding double quotes. The assembler allows for concatenation of strings using the vertical bar. Any number of strings can be concatenated and futhermore can extend over a number of lines.

2. Stack Instructions

load	load stack variable on top of stack
loadi	load immediate (load constant on top of stack)
move(p)	move top of stack into stack variable (and pop stack)
or	bitwise or
and	bitwise and
add	two's complement add
cmp	compare and push a -1, 0 or 1
ashift	arithmetic shift
cshift	circular shift
neg	two's complement negation
com	bitwise complement
pop	pop levels from stack
push	push levels onto stack

LOAD -- load stack variable
004

Description: Loads on top of the stack the contents of the stack variable whose address field is in the byte following the opcode.
sp := sp + 1

Examples:

```
                    0                    .globl  funv
                          procedure      main()
0       1       000     .               .alloc  x
                000
02      2       004     .               load    .
                010
04      3       004     .               load    x
                031
06      4       004     .               load    x
                032
010     5       004     .               load    .-1
                001
012     6       004     .               load    funv
                377
```

Assembler Processing: The stack variable can be any stack level expression.

LOADI -- load immediate
006 or 007

Description: (006) the byte following the opcode is an 8 bit 2's complement number which is converted into an equivalent 16 bit 2's complement number and loaded onto the stack.
sp := sp + 1

(007) the two bytes following the opcode forms a 16 bit 2's complement number. The low order byte is in the first byte following the opcode.
sp := sp + 1

Examples:

```
0       1       000     .               loadi   12      /Octal constant
                012
02      2       006     .               loadi   010     /Octal constant
                010
```

```
04      3       006             loadi   -1
                377
06      4       007             loadi   074001  /*note use of 007
                074001
011     5       007             loadi   32000   /*note use of 007
                076400
```

Assembler Processing: The assembler scans for a number, either octal or decimal, after the LOADI mnemonic. If the number lies between -128 and 127, the shorter form of the instruction is generated. Otherwise, the low bit of the opcode is set indicating that the following two bytes constitute the number.

MOVE, MOVEP -- move to stack variable, move to variable and pop
010 or 011

Description: MOVE (010) The contents of the top of the stack is loaded into the stack variable whose address field is in the byte following the opcode.
sp := sp

MOVEP (011) Same as 010 except that the stack is popped after the move.
sp := sp - 1

Examples:

```
                                dclglob fnum
        0               procedure       main()
0       1       006             dclloc  x
                000
02      2       006             dclloc  y
                000
04      3       006             dclloc  z
                000
06      4       006             loadi   1       /* load constant
                001
010     4       010             move    x       /* move into x
                003
012     4       010             move    fnum    /* move into global
                377
014     4       010             move    y-1     /* move into x
                003
016     4       010             move    .       /* move top into top
                000
020     4       010             move    .-1     /* move top into
                001
                                                /*  second level
```



```
022     3       011             movep   x       /* move and pop
                003
024     2       011             movep   .-1     /* move top into
                001
                                                /*  second level and
                                                /*  pop stack
```

Assembler Processing: Any legal stack variable expression is accepted.

OR -- bitwise or
012 or 013

Description: (012) the byte following the opcode contains the stack address of a variable which is bitwise ored with the contents of the top of the stack. The result is stored in the top level of the stack.
sp := sp

(013) the top two levels of the stack are pooped, bitwise ored, and then pushed back on the stack.
sp := sp - 1

Examples:

```
                                dclglob fnum
        0               procedure       main()
0       1       006             dclloc  x
                000
02      2       006             loadi   07      /* load constant 07
                007
04      2       012             or      x       /* or 07 with x
                001
06      2       012             or      fnum    /* or fnum with result
                377
010     1       013             or              /* or top 2 levels
011     2       007'
                000377
014     1       013             or      0377    /* or top with 0377
```

The effect of the statements at locations 4 and 6 is to or the operand with the top level of the stack and store the result in the top level. The OR instruction at location 010 is without an operand. The effect of the instruction is to pop the top two levels of the stack, or them together, and push the result on top of the stack.

Assembler Processing: The last example of the OR instruction demonstrates the assembler automatically generating the load for the literal constant 0377. The opcode assembled for the OR is 013 which pops the top two levels, ors them, and pushes the result. The net effect is to or the top of the stack with 0377.

AND -- bitwise and
014 or 015

Description: (014) the byte following the opcode contains the stack address field of a variable which is anded with the contents of the top of the stack. The result is stored in the top level of the stack.
sp := sp

(015) the top two levels of the stack are popped, anded together, and then pushed back on the stack.
sp := sp - 1

Examples:

```
                                    dclglob fnum
          0                 procedure       main()
0         1       006       dclloc  x
                  000
02        2       006       loadi   07      /* load constant 07
                  007
04        2       014       and     x       /* and 07 with x
                  001
06        2       014       and     fnum    /* and fnum with
                                            /* top of stack
                  377
010       1       015       and             /* and top 2 levels
011       2       007'
                  000377
014       1       015       and     0377    /* and top with 0377
```

The operands at locations 04 and 06 are stack level expressions. The effect of these statements is to and the operand with the top level of the stack and store the result in the top level. The AND instruction at location 010 is without an operand. The effect of the instruction is to pop the top two levels of the stack, and them together, and push the result on the top of the stack

Assembler Processing: The last example of the AND instruction format is an example in which the assembler automatically generates the load for the literal constant 0377. The opcode assembled is 015 which pops the top two levels, ands them, and pushes the result. The effect is that the number is anded with the top level of the stack and the result is stored in the top level.

ADD -- 16 bit 2's complement add
016 or 017

Description: (016) the byte following the opcode contains the stack address field of a variable which is added using 2's complement arithmetic to the contents of the top of the stack. The result is stored in the top level of the stack.
sp := sp

(017) the top two levels of the stack are popped, added together, and then pushed back on the stack.
sp := sp - 1

Examples:

```
                                    dclglob fnum
          0                 procedure       main()
0         1       006       dclloc  x
                  000
02        2       006       loadi   07      /* load constant 07
                  007
04        2       016       add     x       /* add 07 with x
                  001
06        2       016       add     fnum    /* add fnum with
                  377                       /*  top of stack
010       1       017       add             /* add top 2 levels
011       2       007'
                  000377
014       1       017       add     0377    /* add top to 0377
```

The operands at locations 04 and 06 are stack level expressions. The effect of both statements is to add the operand to the top level of the stack and store the result in the top level. The ADD instruction at location 010 is without an operand. The instruction without an operand pops the top two levels, adds them together, and then pushes the result on the top of the stack.

Assembler Processing: The last example of the ADD instruction demonstrates the assembler automatically generating the load for the literal constant 0377. The opcode assembled for the ADD is 017 which pops the top two levels, adds them, and pushes the result. The net effect of the ADD num statement is that the number is simply added to the top level of the stack.

CMP -- arithmetic compare
020 or 021

Description: (020) the value in the top of the stack is compared to the operand whose address field is the byte following the opcode. Both values are assumed to be 2's complement numbers. If the value in the top of the stack is greater than the operand, a 1 is pushed. If both values are equal a zero is pushed and if the top of the stack is less in value than the operand, a -1 is pushed.
sp := sp + 1

(021) the top two levels of the stack are compared. The resulting 1, 0, or -1 is stored in the top level of the stack.
sp := sp

Examples:

```
                                dclglob fnum
          0                   procedure     main()
0         1         006       dclloc  x
                    000
02        2         004       load    fnum    /* get fnum on top
                    377
04        3         020       cmp     x       /* cmp x and fnum
                    001
                                                /* result on top of
                                                /*  stack
06        4         020       cmp     .       /* one way to load 0
                    000
                                                /* (result is zero)
010       4         021       cmp             /* cmp top two levels

                                /* notice in the next instruction,
                                /* a literal constant load is
                                /* generated. (top of stack before
                                /* the load is either -1, 0, or 1
011       5         006'
                    001
013       5         021       cmp     1       /*result either 0 or
                                                /* -1
```

The effect of the statement 04 is to compare the top of the stack with the operand 'x' and push the result on top of the stack. The CMP instruction at location 010 is without an operand, and as such compares the top two levels and then loads the result into the top level of the stack.

Assembler Processing: The last example of the CMP instruction demonstrates an assembler generated load for the literal constant 1. The opcode assembled for the CMP is 021 which compares the top two levels and loads the result into the top level. The net effect of the CMP num statement is that the value of num is compared to the top of the stack. If greater, a 1 is pushed. If equal, a zero is pushed and if the number is less than the top of the stack, a -1 is pushed.

ASHIFT -- arithmetic shift
024 or 025

Description: (024) the top of the stack is shifted arithmetically the number of bit positions specified in the byte following the opcode. The byte following the opcode is a two's complement number and if negative, the top of the stack is shifted right with sign extension (high order bit replicated right). If the byte contains a positive number, the top of the stack is shifted left with zero's shifted in on the right. In other words if the value of the byte following the opcode is n, the effect is to multiply the top of the stack by 2**n.

For example, suppose the top of the stack contained 040 base 8. If the value in the byte following the opcode were 01, then after execution, the top would contain 0100. On the other hand, if the byte following the opcode contained -3 (0375), the top of the stack would be shifted right three places yielding 004. Now suppose that the top of the stack contains a negative 8 (0370). If the byte following the instruction contained a -2 (0376), after execution, the top would contain -2 (0376).
sp := sp

(025) identical to (024) except the arithmetic shift value (16 bit two's complement in this case) is popped from the stack. For example, if the top of the stack contained 03 and the second level contained 007, after execution the stack pointer would be one level lower and the new top of the stack would contain 070.
sp := sp - 1

Examples:

```
0         1         006       loadi   040     /*load 040 on stack
                    040
02        1         024       ashift  01      /* shift left by 1
                    001
                                                /* top now = 0100
04        1         024       ashift -2       /* shift right by 2
                    376
                                                /* top now 020
06        2         006       loadi   -3      /* load negative
                    375
010       2         024       ashift  1       /* mult by 2**1
                    001
                                                /* top now = -6 (0372)
```

Assembler Processing: The last example of the OR instruction demonstrates the assembler automatically generating the load for the literal constant 0377. The opcode assembled for the OR is 013 which pops the top two levels, ors them, and pushes the result. The net effect is to or the top of the stack with 0377.

AND -- bitwise and
014 or 015

Description: (014) the byte following the opcode contains the stack address field of a variable which is anded with the contents of the top of the stack. The result is stored in the top level of the stack.
sp := sp

(015) the top two levels of the stack are popped, anded together, and then pushed back on the stack.
sp := sp - 1

Examples:

```
                                    dclglob fnum
          0          procedure       main()
0         1    006      dclloc     x
               000
02        2    006      loadi      07        /* load constant 07
               007
04        2    014      and        x         /* and 07 with x
               001
06        2    014      and        fnum      /* and fnum with
                                               /* top of stack
               377
010       1    015      and                  /* and top 2 levels
011       2    007'
               000377
014       1    015      and        0377      /* and top with 0377
```

The operands at locations 04 and 06 are stack level expressions. The effect of these statements is to and the operand with the top level of the stack and store the result in the top level. The AND instruction at location 010 is without an operand. The effect of the instruction is to pop the top two levels of the stack, and them together, and push the result on the top of the stack

Assembler Processing: The last example of the AND instruction format is an example in which the assembler automatically generates the load for the literal constant 0377. The opcode assembled is 015 which pops the top two levels, ands them, and pushes the result. The effect is that the number is anded with the top level of the stack and the result is stored in the top level.

ADD -- 16 bit 2's complement add
016 or 017

Description: (016) the byte following the opcode contains the stack address field of a variable which is added using 2's complement arithmetic to the contents of the top of the stack. The result is stored in the top level of the stack.
sp := sp

(017) the top two levels of the stack are popped, added together, and then pushed back on the stack.
sp := sp - 1

Examples:

```
                                    dclglob fnum
          0          procedure       main()
0         1    006      dclloc     x
               000
02        2    006      loadi      07        /* load constant 07
               007
04        2    016      add        x         /* add 07 with x
               001
06        2    016      add        fnum      /* add fnum with
               377                           /*  top of stack
010       1    017      add                  /* add top 2 levels
011       2    007'
               000377
014       1    017      add        0377      /* add top to 0377
```

The operands at locations 04 and 06 are stack level expressions. The effect of both statements is to add the operand to the top level of the stack and store the result in the top level. The ADD instruction at location 010 is without an operand. The instruction without an operand pops the top two levels, adds them together, and then pushes the result on the top of the stack.

Assembler Processing: The last example of the ADD instruction demonstrates the assembler automatically generating the load for the literal constant 0377. The opcode assembled for the ADD is 017 which pops the top two levels, adds them, and pushes the result. The net effect of the ADD num statement is that the number is simply added to the top level of the stack.

CMP -- arithmetic compare
020 or 021

Description: (020) the value in the top of the stack is compared to the operand whose address field is the byte following the opcode. Both values are assumed to be 2's complement numbers. If the value in the top of the stack is greater than the operand, a 1 is pushed. If both values are equal a zero is pushed and if the top of the stack is less in value than the operand, a -1 is pushed.
sp := sp + 1

(021) the top two levels of the stack are compared. The resulting 1, 0, or -1 is stored in the top level of the stack.
sp := sp

Examples:

```
                                dclglob fnum
          0             procedure       main()
0         1     006               dclloc  x
                000
02        2     004               load    fnum    /* get fnum on top
                377
04        3     020               cmp     x       /* cmp x and fnum
                001
                                                  /* result on top of
                                                  /*  stack
06        4     020               cmp     .       /* one way to load 0
                000
                                                  /* (result is zero)
010       4     021               cmp             /* cmp top two levels

                                  /* notice in the next instruction,
                                  /* a literal constant load is
                                  /* generated. (top of stack before
                                  /* the load is either -1, 0, or 1
011       5     006'
                001
013       5     021               cmp     1       /*result either 0 or
                                                  /* -1
```

The effect of the statement 04 is to compare the top of the stack with the operand 'x' and push the result on top of the stack. The CMP instruction at location 010 is without an operand, and as such compares the top two levels and then loads the result into the top level of the stack.

Assembler Processing: The last example of the CMP instruction demonstrates an assembler generated load for the literal constant 1. The opcode assembled for the CMP is 021 which compares the top two levels and loads the result into the top level. The net effect of the CMP num statement is that the value of num is compared to the top of the stack. If greater, a 1 is pushed, If equal, a zero is pushed and if the number is less than the top of the stack, a -1 is pushed.

ASHIFT -- arithmetic shift
024 or 025

Description: (024) the top of the stack is shifted arithmetically the number of bit positions specified in the byte following the opcode. The byte following the opcode is a two's complement number and if negative, the top of the stack is shifted right with sign extension (high order bit replicated right). If the byte contains a positive number, the top of the stack is shifted left with zero's shifted in on the right. In other words if the value of the byte following the opcode is n, the effect is to multiply the top of the stack by 2**n.

For example, suppose the top of the stack contained 040 base 8. If the value in the byte following the opcode were 01, then after execution, the top would contain 0100. On the other hand, if the byte following the opcode contained -3 (0375), the top of the stack would be shifted right three places yielding 004. Now suppose that the top of the stack contains a negative 8 (0370). If the byte following the instruction contained a -2 (0376), after execution, the top would contain -2 (0376).
sp := sp

(025) identical to (024) except the arithmetic shift value (16 bit two's complement in this case) is popped from the stack. For example, if the top of the stack contained 03 and the second level contained 007, after execution the stack pointer would be one level lower and the new top of the stack would contain 070.
sp := sp - 1

Examples:

```
0         1     006               loadi   040     /*load 040 on stack
                040
02        1     024               ashift  01      /* shift left by 1
                001
                                                  /* top now = 0100
04        1     024               ashift -2       /* shift right by 2
                376
                                                  /* top now 020
06        2     006               loadi   -3      /* load negative
                375
010       2     024               ashift  1       /* mult by 2**1
                001
                                                  /* top now = -6 (0372)
```

```
012     2     024           ashift  -3      /* mult by 2**-3
              375
                                            /* to now -1 (0377)
014     3     006           loadi   2       /* load constant 2
              002
016     4     006           loadi   1       /* load amount to shift
              001
020     3     025           ashift          /* shift 2 by 1 left
                                            /* top = 04
021     4     006           loadi   -2      /* load constant
              376
023     3     025           ashift          /* shift 04 by -2
                                            /* top = 01
```

Assembler Processing: If a numerical argument is given, the number is stored as a byte after the instruction. If no number follows the instruction, the opcode used is 025.

CSHIFT -- 16 bit circular shift
026 or 027

Description: (026) the top of the stack is shifted circularly the number of bit positions specified by the 8 bit two's complement number in the byte following the opcode. If the number is negative, the top of the stack is shifted right, and if positive, the stack is shifted left. When shifting right, the lowest order bit is shifted into the high order bit; the opposite holding true when shifting left.

For example, if the top of the stack contained 055555 octal and the second byte of the instruction contained 001, after the instruction the top of the stack would contain 0133332. The high order bit shifted into the low order bit in this case is zero. On the other hand, suppose the top of stack contained 0100001 (high and low order bits set). If the second byte of the instruction contained -2 (0376), then after execution the top level would contain 060000 (a one and a zero bit were shifted in on the left).
sp := sp

(027) the same as 026 except that the number to shift is popped off of the stack instead of following the opcode. For example, if the top of the stack contained -1 (0177777) and the second level contained 040005, after execution the stack pointer would be one level lower and the new top would contain 0120002.
sp := sp - 1

Examples:


```
0       1     007           loadi   055555  /* load onto stack
              055555
03      1     026           cshift  1       /* shift top level 1
              001
                                            /*  bit position left
                                            /* top now is 0133332

05      2     007           loadi   0100001 /* load 2 on stack
              100001
010     2     026           cshift  -2      /* shift 2 right to
              376
                                            /*  form 060000

012     3     007           loadi   040005  /* load 2 on stack
              040005
015     4     006           loadi   -1      /* load -1 on stack
              377
017     3     027           cshift          /* pop -1 from stack,
                                            /*  shift 040005 right
                                            /*  1 to form 0120002.
```

Assembler Processing: If a numerical argument is given, the number is stored as a byte after the opcode. The number can range from -128 to 127. If no numerical argument is given, the opcode used is 027.

NEG -- negate the top of the stack
134

Description: two's complement negation of the top level of the stack.
sp := sp

Examples:

```
0       1     006           loadi   10      /* load decimal 10
              012
02      1     134           neg             /* negate top of stack
                    /* top of stack now contains -10 (0177776)
```

Assembler Processing: The assembler expects only the mnemonic

COM -- bitwise complement of the top of the stack
136

Description: the top of the stack is complemented bitwise.
sp := sp

Examples:

```
0     1     007          loadi   0512    /* load octal 512
            000512
03    1     136          com             /* complement top
                  /* top of stack now contains 177265 base 8
```

Assembler Processing: The assembler expects only the mnemonic.

POP -- pop levels from the stack
030

Description: the stack is popped the number of levels indicated in the byte following the opcode. The byte following the opcode is taken to be an eight bit positive number [0-255].
sp := sp - #

Examples:

```
0     1     006          dclloc  x
            000
02    2     006          loadi   0       /* increase stack pointer
            000
04    3     006          loadi   1
            001
06    4     006          loadi   2
            002
010   3     030          pop     1       /* pop 1 level from stack
            001

                  /* If a local variable follows the pop instr,
                  /*  the assembler will generate the number
                  /*  of levels to pop so that the level
                  /*  associated with the local variable is
                  /*  the new top of the stack.  For example,

012   1     030          pop     x       /* pops down to level
            002
                                         /*  of variable x
```


Assembler Processing: If a number follows the mnemonic, the value of the number is assembled into the second byte of the instruction. If a local stack variable is given, the assembler computes the displacement from the current level down to the level associated with the variable and assembles that value into the second byte of the instruction.

PUSH -- push levels onto the stack
032

Description: the stack is increased by the number of levels indicated in the byte following the instruction which is assumed to be an 8 bit positive number [0-255].
sp := sp + #

Examples:

```
0     4     032          push    4       /* increse sp by 4
            004
                  x = .                  /* define loc var x
02    1     030          pop     3       /* decrease sp by 3
            003
                  /* If a local variable follows the "PUSH" instr,
                  /*  the assembler will compute the number of
                  /*  levels to push to make the level associ-
                  /*  ated with the variable the new top of
                  /*  stack.  For example,
04    4     032          push    x       /* push levels to make
            003
                                         /*  x new top of stack.
```

Assembler Processing: If a number follows the instruction, the value of the number is assembled after the opcode. If a local stack variable is given as an operand, the assembler calculates the displacement from the current stack pointer to the level associated with the variable and assembles that value after the opcode. The resulting instruction forces the stack pointer at the level of the local variable.

3. Control Transfer Instructions:

br(p)	unconditional branch (and pop stack)
beq(p)	branch if equal to zero (and pop stack)
bne(p)	branch not equal to zero (and pop stack)
bge(p)	branch greater than or equal to zero (and pop stack)
bgt(p)	branch greater than zero (and pop stack)
ble(p)	branch less than or equal to zero (and pop stack)
blt(p)	branch less than zero (and pop stack)
jmp	unconditional jump (anywhere in current procedure)
jmptble	jump indirecty through an address table
ibrz	increment and branch on zero
dbrz	decrement and branch on zero
testbr	test and branch if equal

BR, BRP -- unconditional branch, unconditional branch with pop
054, 055

Description: (054) the contents of the byte following the opcode is fetched and added to the value of the program counter plus one. The resulting 16 bit address is loaded into the program counter transfering program control to the new address. The 8 bit offset allows for branching 128 locations forward or 127 locations backward.
sp := sp

(055) same as 054 except that the stack is popped.
sp := sp - 1

Examples:
A few words need to be said about the stack value and branching. Because local variables are addressed relative to the top of the stack, the stack pointer must be unique at each instruction for local variables to have unique addresses. The GAS assembler automatically checks for stack agreement of all references to a particular label. The value associated with a label is the value of the stack at the time the label is first encountered either as a label prefix or as a branch address. If the levels do not agree, the assembler will issue an appropriate warning message.

The value of the stack after an unconditional branch or jump is ambiguous unless the following label already has a stack level associated with it. In the cases in which the label has not been encountered previously and does not have a value associated with it, the assembler uses the current value of the stack or in other words, the value of the stack after the unconditional branch. A pseudo op ". = " is available to alter the assumed stack level after an unconditional branch, jump or return statement.

In the following examples, the assembly listing has been edited to fill in the correct forward address reference (ie as in the branch to "done"). In a normal listing, all forward address references are shown as zeros.

```
    0     1     000          .dalloc  x
                000
02    1     054  000  start:  br       done    /* forward reference..
                001
                                              " address in
                             /* The stack pointer is ambiguous at this
                             /* point.  The ". =" pseudo op is used to
                             /* change the assumed stack value.
          0                  . = .-1
04    1     006  000  label:  loadl   1
                001
                             /* The stack value associated with "label" is
                             /* 0, while the stack value associated with
                             /* "done" is 1.
```

```
06      0       055     done:   brp     label   /* branch and pop
                374

                        /* Now for some common errors.  The stack is
                        /* presently assumed to be 0.  The stack
                        /* value associated with "start" is 1.
warning--stack level is ambiguous
stack level conflict for label 'start:' first encountered at pc=02
010     0       054     15:     br      start   /* note error
                370
                        /* Correct the problem with pseudo op
                        /* by setting sp to level associated with x
        1               . = x
012     1       054     16:     br      start
                366
```

A final programming remark: It is generally possible to program such that the value associated with a label is not ambiguous. This can be accomplished by placing labeled sections of code which are not "in line" after any refence to the label.

Assembler Processing: The symbol following the BR or BRP mnemonics is assumed to be a label identifier. If the label id is not in the symbol table, it is entered and assigned the value of the stack after execution of the instruction. If the label is in the symbol table, the value of the stack pointer after execution of the instruction is compared to the value in the symbol table. If these values disagree, an error message is printed. Furthermore, if the label has been defined (ie the label prefix has been scanned), the offset to the label is checked for being greater than -128 and is assembled as a byte after the opcode. If the label is undefined, a zero byte is assembled into the object file and an entry is made in the forward reference list so that the displacement can be filled at the end of the program. The ambiguous flag is set indicating that unless a defined label follows, the stack pointer is ambiguous.

BEQ, BEQP, BNE, BNEP, BGE, BGEP, BGT, BGTP, BLE, BLEP, BLT, BLTP, -- branch if equal to, not equal to, greater than or equal to, greater than, less than or equal to, less than zero
056 to 071

Description: conditional branch instructions. If the condition is true for the value on top of the stack, control is transfered to the address whose offset is contained in the byte following the opcode. The absolute address is formed by adding the 8 bits 2's complement offset to the 16 bit absolute address containing the offset, plus one. This scheme allows for branching 123 locations foreward and 127 locations backward.

BEQ (056) branch if equal to zero
sp := sp

BEQP (057) branch if equal to zero and pop stack.
sp := sp - 1

BNE (060) branch if not equal to zero
sp := sp

BNEP (061) branch if not equal to zero and pop stack
sp := sp - 1

BGE (062) branch if greater than or equal to zero
sp := sp

BGEP (063) branch if greater than or equal to zero and pop stack
sp := sp - 1

BGT (064) branch if greater than zero
sp := sp

BGTP (065) branch if greater than zero and pop stack
sp := sp - 1

BLE (066) branch if less than or equal to zero
sp := sp

BLEP (067) branch if less than or equal to zero and pop stack
sp := sp - 1

BLT (070) branch if less than zero
sp := sp

BLTP (071) branch if less than zero and pop stack
sp := sp - 1

Examples:
The foreward branch displacements have been edited into the following listing:

```
0       1       006             delloc  x       /* local variable x
                000
02      2       004             load    x
                000
04      2       070     start:  blt     done    /* branch to done if
                014
                                                /*  top of stack .LT.
                                                /*  zero.
06      1       065             bgtp    done2  /* brach to done2 if
                016
                                                /*  top of stack .GT.
```

```
                                        /*  0 and pop stack
010     2       004             load    x
                000
012     2       056             beq     done    /* branch if .EQ. 0
                006
014     1       063             bgep    done2   /* branch if .GE. 0
                010
                                        /*  and pop stack
016     2       004             load    x
                000
020     2       060             bne     start   /* branch if non zero
                362
022     1       067     done:   blep    done2   /* branch less than or
                002
                                        /*  equal to zero and
                                        /*  pop stack
024     1       010             move    y       /* move top of stack
                000
                                        /*  into x
026     2       006     done2:  loadi   10      /* load constant 10..
                012
```

Assembler Processing: The symbol following the instruction mneronic is assumed to be a label identifier. If an entry in the symbol table exists for the label (ie backward reference), the displacement is checked for being in the range [-128,0] and assembled into the byte following the opcode. An error will result if the stack level associated with the label does not match the stack level at the conclusion of the branch instruction.

If an entry does not exist for the label identifier, an entry is created and given the attributes LAB_UDEF and LOCALSCOPE and the stack level associated with the foreward label is the stack level at the conclusion of the instruction (ie if there is a nop, the level assigned to the label is the level before the branch instruction minus 1). A null byte is assembled after the instruction for the yet unkown displacement. An entry is created in the foreward reference list so that the correct displacement can be filled in at the end of pass 1.

JMP -- unconditional jump
072

Description: the contents of the two bytes following the opcode are combined to form a 16 bit address which is loaded into the program counter. The low order 8 bits of the address are in the

first byte following the opcode. The 16 bit absolute address allows for branching to any location in the 65 K byte program space.
sp := sp

Examples:
The forward jump addresses have been edited into the following program listing. In a normal listing, the forward address references would be zero.

```
0       1       006             dclloc  x
                000
02      2       006     start:  loadi   1
                001
04      1       030             pop     1
                001
06      1       072             jmp     start
                000002

                        /* Since the following statement label "lab12"
                        /*  has not been encountered and hence does
                        /*  not have a stack value associated with it,
                        /*  the stack pointer is ambiguous.
warning--stack level is ambiguous
011     1       072     lab12:  jmp     lab13
                000017

                        /* "lab13" has not been encountered. Once
                        /*  the stack pointer is ambiguous.  I have
                        /*  used the ". =" pseudo op to set the level
                        /*  of the stack and remove the ambiguity.
        1                       . = .
014     1       072     lab13:  jmp     start
                000002

                        /* "lab14" has been encountered so the stack
                        /*  will not be ambiguous.
017     1       072     lab14:  jmp     start
                000002
```

Assembler Processing: The processing is identical to that of the JP instruction except that no limitations are placed on the address of the label. This is due to the fact that two bytes are available for the address instead of one.

JMPTBLE -- pop stack and index into address table for branch address
074

Description: the value on top of the stack is popped and used to index into a table of 16 bit addresses which follow the opcode. The index of the first address is 1. If the value is less than or equal to zero or greater than the number of addresses in the table, control passes to the following instruction. The first byte of the address table contains the number of bytes used to store the table (twice the number of addresses).
sp := sp - 1

Examples:
The forward addresses have been edited into the following program listing.

```
0     1   006        .       dclloc  x
          000
02    1   010        11:     move    x       /* dummy instruction
          000
04    1   010        12:     move    x
          000
06    1   010        13:     move    x
          000

010   2   004        doit:   load    x
          000
                     gotoit: jmptble 11, 12 13,
                                     14 15,  /* comment
012   1   074                         16
          014
          000002
          000004
          000006
          000030                             /* these addresses
          000032                             /*  have been edited
          000034                             /*  in
030   1   054                 br      doit    /* default
          356

032   1   010        14:     move    x       /* dummy instruction
          000
034   2   006                 loadi   10
          012
036   2   054                 br      gotoit
          352

040   1   010        15:     move    x
          000
042   1   010        16:     move    x
          000
```


Assembler Processing: The symbols following the JMPTBLE mnemonic are assumed to be label identifiers and are treated as address references. That is to say, if the label is in the symbol table, the value of the stack associated with the label is compared to the value of the stack after the JMPTBLE instruction. If unequal, an error message is listed. If the label is not in the symbol table, an entry is created and the current value of the stack is stored in the entry for future comparison. In addition, an entry is created in the forward address reference list so that the address of the label can be filled in at the conclusion of pass one.

The stack is not considered ambiguous after the instruction since control can pass to the following statement. If a comma is the last token in the line or last token before a comment, CAS assumes that the instruction continues on the next line.

IBRZ, DBRZ -- increment (decrement) and branch on zero
022 or 023

Description: (022) the variable whose address field lies in the byte following the opcode is incremented. If the result is zero, control passes to the address specified in the next two bytes (a standard word address). The total length of the instruction is 4 bytes.
sp := sp

(023) the same as 022 except that the variable is decremented by one.
sp := sp

Examples:

```
0     1   006                 dclloc  x
          000
02    1   054        done:    br      loop
          006
04    1   010        loop:    move    .       /*effective no-op
          000
06    1   022                 ibrz    x, done /*inc top of stack
          000
          000002
                                             /*and branch on zero
                                             /* to done.
012   1   022                 ibrz    .-1, done
          001
          000002
016   1   023                 ibrz    x, done /* decrement second
```

```
001
000002
                                    /* level and branch
                                    /* to done if zero
```

Assembler Processing: The stack level expression is delineated from the branch address by a comma. This is one of the few instructions in which a comma is necessary. Note that the branch address is a full two bytes (16 bits) long which allows for branching to any program address.

TESTBR -- compare top of stack with byte following opcode and branch to address if equal

076

Description: the byte following the instruction is converted to a 16 bit number by extending the high order bit to the left and then compared with the value popped off the stack. If equal, control passes to the 16 bit absolute address specified in the next two bytes. If not equal, control passes to the instruction following the TESTBR instruction. Note that any label in the program space can be addressed by the 16 bits. The byte following the opcode is an 8 bit two's complement number capable of representing -128 to 127.
sp := sp - 1

Examples:

```
0       1       006             dclloc  x
                000
02      2       004     lab1:   load    x       /* load x on stack
                000
04      3       004             load    .       /* load x again
                000
06      2       076             testbr  10,  lab2
                012
                000000
                        /* Note that in the above statement
                        /*  the address of lab2 is not filled in
                        /*  in the assembly listing because the
                        /*  address is not known at this point
                        /*  in the assembly.

012     1       076     lab2:   testbr  -2   lab1
                376
                000002
```


Assembler Processing: The assembler first scans for a number after the instruction mnemonic. The number is checked for lying in the range [-128,127] and then is assembled as a byte after the opcode. The assembler then scans for an identifier which is assumed to be a label. If the label is in the symbol table, the the stack level associated with the label is compared with the stack level after the instruction has executed. If the label has been defined, the 16 bit absolute address is assembled after the opcode and the byte containing the number. If the label is not in the symbol table, indicating that this is the first reference to the label, an entry is created for the label having the attribute UNDEFINED and the stack value associated with the label is the value of the stack after execution of this instruction.

4. Text Buffer Instructions:

tbappc	append character to text buffer
tbappt	append text string to text buffer
tbinsc	insert character in text buffer
tbinst	insert text string in text buffer
tbdelc	delete character in text buffer
tbdelt	delete text in text buffer
tbchngc	change character in text buffer
tbpshc	push character in text buffer onto stack
tbcmp	compare text string with string in text buffer
tbsearch	search text buffer for a character string
tbcnvo	convert value to an octal character representation
tbcnvd	convert value to a decimal character representation
tbinit	initialize text buffer to a null string
tbmp1	move text pointer p1
tbmp2	move text pointer p2
tbldp1	load value of text pointer p1 on stack
tbldp2	load value of text pointer p2 on stack
tbldln	load number of characters in text buffer on stack
tbstp1	set value of text pointer p1
tbstp2	set value of text pointer p2

TBAPPC -- append character to text buffer
152

Description: the value on top of the stack is popped and the low order 8 bits are appended to the text buffer as a character. "p1" is unchanged. If "p2" is equal to the number of characters in the buffer ("ln"), "p2" is incremented by one. "ln" is incremented by one in either case. The text buffer will hold a maximum of 80 characters. If an attempt is made to append to a full buffer, the character is flushed and the pointers are unchanged.
sp := sp - 1

Examples:

```
0      0     140          tbinit
01     1     006          loadi   0141    /* load octal for 'a'
                141
03     2     152          tbappc
                      /* TEXT BUFFER: a
                      /*              ↑↑
                      /*             p1p2
                      /* p1 = 0     p2 = ln = 1

04     3     006          loadi   0142    /* load octal for 'b'
                142
06     4     152          tbappc
                      /* TEXT BUFFER: ab
                      /*              ↑ ↑
                      /*             p1 p2
                      /* p1 = 0     p2 = ln = 2

07     4     044          tbmp2   -1      /* move p2 left 1 pos
                377
                      /* TEXT BUFFER: ab
                      /*              ↑↑
                      /*             p1p2
                      /* p1 = 0     p2 = 1     ln = 2

011    5     006          loadi   0143    /* load octal for 'c'
                143
013    6     152          tbappc
                      /* TEXT BUFFER: abc
                      /*              ↑↑
                      /*             p1p2
                      /* p1 = 0     p2 = 1     ln = 3
```

Assembler Processing: None

TBAPPT -- append text to text buffer
034 or 035

Description: (034) the text group following the opcode is appended to the text buffer. The length of the text buffer "ln" is increased by the number of characters in the string and if "p2" ≥ "p1", "p2" is increased by the number of characters.

A text group consists of a byte containing the character count followed by the characters packed one to a byte. The text group representing the string "abc" would be the following 4 bytes:

```
003     /*character count
141     /* Octal ASCII 'a'
142     /* 'b'
143     /* 'c'
```

The length in bytes of a text group is the number of characters plus one.
sp := sp

(035) the text in the current text line of the current file is appended to the text buffer. "ln" and "p2" are treated as above.
sp := sp

Examples:

```
0     0     140          tbinit          /* set p1=p2=ln=0
01    0     034          tbappt "abc"    /* append "abc" to
            003
            141
            142
            143
                                         /*  text buffer.
                  /* TEXT BUFFER: abc
                  /*              ↑  ↑
                  /*             p1  p2
                  /*  p1 = 0      p2 = ln = 3
                  /*
06    0     034          tbappt  "defg"  /* append "defg"
            004
            144
            145
            146
            147
                  /* TEXT BUFFER: abcdefg
                  /*              ↑      ↑
                  /*             p1      p2
                  /*  p1 = 0      p2 = ln = 7
                  /*
                  /* suppose current text line is "xyz".  Then
                  /*  TBAPPT without an argument yields:
014   0     035          tbappt          /* append from cur tln
                  /* TEXT BUFFER: abcdefgxyz
                  /*              ↑      ↑
                  /*             p1      p2
                  /* p1 = 0      p2 = ln = 10
                  /*
                  /*
                  /* Now an example of a concatenated string:
                        tbappt  " willie"  |   /* comment
015   0     034                  " wonka"
            015
            040
            167
            151
            154
            154
            151
            145
            040
            167
            157
            156
            153
            141
                  /* TEXT BUFFER: abcdefgxyz willie wonka
                  /*              ↑                      ↑
                  /*             p1                     p2
                  /* p1 = 0      p2 = ln = 23
```


Assembler Processing: If a text string follows the instruction mnemonic, it is assembled as a standard text group after the instruction. If no string is given, the opcode assembled is 035. Note use of concatenate symbol '|'. Two strings can be concatenated on same line but everything after a '/*' is considered part of the comment.

TBINSC -- insert character before "p1" in text buffer
144

Description: the value on top of the stack is popped and the low order 8 bits are inserted in the text buffer as a character before "p1". The characters from "p1" to the end are shifted right one character position. "ln" is incremented by 1. "p1" is unchanged. "p2" is incremented by one if greater than "p1".
sp := sp - 1

Examples:
The following example was assembled with the "-s" option which

abbreviates the octal text expansion in the listing.

```
0     0     140          tbinit
01    0     034          tbappt  "sample text"
                 013
                 163
016   0     042          tbmpl   2
                 002
                    /* TEXT BUFFER: sample text
                    /*                ↑        ↑
                    /*               p1       p2
                    /* p1 = 2     p2 = ln = 11
020   1     006          loadi   0101    /* load octal for 'A'
                 101
023   0     144          tbinsc          /* insert 'A' b4 p1
                    /* TEXT BUFFER: saAmple text
                    /*                ↑         ↑
                    /*               p1        p2
                    /* p1 = 2     p2 = ln = 11
```

Assembler Processing: None

TBINST -- insert text in text buffer before p1
036 or 037

Description: (036) the text group following the instruction is inserted before "p1" in the text buffer. The length of the text buffer "ln" is increased by the number of characters in the string and if "p2" >= "p1", p2 increased by the number of characters. "p1" is unaffected.
sp := sp

(037) the text in the current text line of the current active file is inserted before p1 in the text buffer. "p1", "p2" and "ln" are treated as above.
sp := sp

Examples:

```
0     0     034          tbappt  "abc"
                 003
                 141
                 142
                 143
```



```
                    /* TEXT BUFFER: abc
                    /*               ↑  ↑
                    /*              p1  p2
                    /* p1 = 0       p2 = ln = 3
05    0     036          tbinst  "xyz"
                 003
                 170
                 171
                 172
                    /* TEXT BUFFER: xyzabc
                    /*               ↑     ↑
                    /*              p1    p2
                    /* p1 = 0       p2 = ln = 6
012   0     042          tbmpl   1       /* move p1 1 char
                 001
                                         /*  position right
                    /* TEXT BUFFER: xyzabc
                    /*                ↑    ↑
                    /*               p1   p2
                    /* p1 = 1        p2 = ln = 6
                    /*
                    /* Suppose that the current text line contains
                    /*   "TERMINAL "
014   0     037          tbinst
                    /* TEXT BUFFER: xTERMINAL yzabc
                    /*                ↑            ↑
                    /*               p1           p2
                    /* p1 = 1        p2 = ln = 15
```

Assembler Processing: If a text string follows the instruction mnemonic, it is assembled as a standard text group after the instruction opcode. If no string is given, 037 is assembled.

TBDELC -- delete character at "p1" and compress text
146

Description: the character at "p1" in the text buffer is deleted and the remainder of the line is shifted left one position. "p1" is unchanged. If "p2" >"p1", "p2" is decremented by 1. The number of characters in the text buffer "ln" is decremented.
sp := sp

Examples:
The following example was assembled using the "-s" option which abbreviates the expansion in the listing of text in octal.

```
0     0     140          tbinit
01    0     034          tbappt  "sample text"
                 013
                 163
016   0     042          tbmpl   2
                 002
                    /* TEXT BUFFER: sample text
                    /*                ↑       ↑
                    /*               p1      p2
                    /* p1 = 2     p2 = ln = 11
020   0     146          tbdelc          /* delete char at p1
                    /* TEXT BUFFER: saple text
                    /*                ↑       ↑
                    /*               p1      p2
                    /* p1 = 2     p2 = ln = 10
```

Assembler Processing: None

TBDELT -- delete text in text buffer between "p1" and "p2"
154

Description: the characters from "p2" to the end of the buffer are moved left to "p1" deleting the text from "p1" up to but not including "p2". "p2" is set to "p1" and the text buffer length "ln" is changed to reflect the new length of the text buffer. An error will result if "p2" is less than "p1".
sp := sp

Examples:
The expansion of the text string in octal has been abbreviated in the following example:

```
0     0     140          tbinit
01    0     034          tbappt  "sample text"
                 013
                 163
016   0     042          tbmpl   2       /* move p1 right 2
                 002
020   0     044          tbmp2   -1      /* move p2 left 1 char
                 377
                    /* TEXT BUFFER: sample text
                    /*                ↑       ↑
                    /*               p1      p2
                    /* p1 = 2     p2 = 10    ln = 11
022   0     154          tbdelt          /* delete text
```



```
                    /* TEXT BUFFER: sat
                    /*                ↑
                    /*               p1,p2
                    /* p1 = p2 = 2      ln = 3
```

Assembler Processing: None

TBCHNGC -- change character at "p1"
150

Description: the value on the stack is popped and the low order 8 bits are stored at "p1" in the text buffer. "p1", "p2", and "ln" are unchanged.
sp := sp - 1

Examples:

```
0     0     140          tbinit
01    0     034          tbappt  "sample text"
                 013
                 163
016   0     042          tbmpl   2
                 002
                    /* TEXT BUFFER: sample text
                    /*                ↑       ↑
                    /*               p1      p2
                    /* p1 = 2     p2 = ln = 11
020   1     006          loadi   0115    /* load octal '%'
                 115
023   0     150          tbchngc         /* store at p1
                    /* TEXT BUFFER: sa%ple text
                    /*                ↑       ↑
                    /*               p1      p2
                    /* p1 = 2     p2 = ln = 11
```

Assembler Processing: None

TBPSHC -- push character at "p1" on stack
142

Description: the 8 bit ascii character at "p1" in the text buffer is pushed onto the stack.
sp := sp + 1

Examples:

```
0     0    140          tbinit
01    0    034          tbappt  "sample text"
                013
                163
                141
                155
                160
                154
                145
                040
                164
                145
                170
                164
                     /* TEXT BUFFER: sample text
                     /*              ↑          ↑
                     /*             p1         p2
                     /* p1 = 0     p2 = ln = 11
016   1    142          tbpshc          /* push 's' on stack

                     /* top of stack contains ascii 's' =  0163

017   1    042          tbmpl   2
                002
                     /* TEXT BUFFER: sample text
                     /*               ↑        ↑
                     /*              p1       p2
                     /* p1 = 2     p2 = ln = 11
021   2    142          tbpshc          /* push 'm' on stack

                     /* top of stack contains 'm' =  0155
```

Assembler Processing: None

TBCMP -- compare text with that in text buffer
040 or 041

Description: (040) the text group following the opcode is compared lexically with the text between "p1" and "p2" in the text buffer. If the operand text string (the text group) is greater than the string between "p1" and "p1", a 1 is loaded onto the stack. If equal, a zero is pushed onto the stack and if less, a -1 is pushed. The lexical comparison is based on the ASCII codes for the symbols. Characters have the following ordering:
blank < 0 < 9 < A < Z < a < z

A user error results if "p2" < "p1".
sp := sp + 1

(041) same as above except the current text line in the current active file is used in place of the text group following the opcode.
sp := sp + 1

Examples:

```
0     0    034          tbappt  "cap"
                003
                143
                141
                160
                     /* TEXT BUFFER: cap
                     /*              ↑  ↑
                     /*             p1  p2
                     /* p1 = 0       p2 = ln = 3
05    1    040          tbcmp   "capital"
                007
                143
                141
                160
                151
                164
                141
                154
                     /* Operand string "capital" is lexically
                     /*  greater than "cap" therefore a 1 is
                     /*  loaded onto the stack.
                     /* Suppose that the current text line of
                     /*  the current active file contained
                     /*  "ca".  The following instruction
                     /*  would load a -1 onto the stack.
016   2    041          tbcmp           /* "ca" < "cap"

                     /* Note that string in the text buffer
                     /*  "between p1 and p2" are the characters
                     /*  from p1 up to but not including char-
                     /*  acter at p2.
017   2    042          tbmpl   1       /* move p1 to right
```

```
                001
                     /* TEXT BUFFER: cap
                     /*               ↑ ↑
                     /*              p1 p2
                     /* p1 = 1       p2 = ln = 3
    021     3   040          tbcmp   "caa"   /* "caa" > "ap"
                003
                143
                141
                141
                                             /* top of stack = 1
```

Assembler Processing: If a text string follows the instruction mnemonic, it is assembled as a standard text group after the opcode. If no text is given, the opcode assembled is 041.

TSEARCH -- scan text buffer for operand text string
046 or 047

Description: (046) the text buffer between "p1" and "p2" is scanned for a string which matches the text string in the standard text group following the opcode. If a match is found, "p1" and "p2" are set to bracket the string and control branches to the absolute address contained in the two bytes following the text group.
sp := sp

(047) the same as 046 except that the current text line in the current active file is used in place of the text group. No text group follows the instruction, only the two bytes which contain the branch address.
sp := sp

Examples:

```
    0       1   006          dclloc  x
                000
    02      2   006  foundit: loadi   1       /* branch here on match
                001
    04      1   030          pop     1       /* dummy operation
                091
                     /* suppose we have
                     /* TEXT BUFFER: save: abc
                     /*               ↑       ↑
                     /*              p1      p2
                     /* p1 = 0       p2 = ln = 9
```



```
                     /* In the following statement, a match will
                     /*  be found, p1 will be set to 0, p2 will
                     /*  set to 5, and control will pass to
                     /*  "foundit".
    06      1   046          tsearch "save:", foundit
                005
                163
                141
                166
                145
                072
                000002
                     /* Suppose the text buffer is as above and
                     /*  and the current text line in the current
                     /*  active file is:
                     /*     "abc"
                     /*  then the following instruction will
                     /*  find a match, set p1 to 6, p2 to 9 and
                     /*  transfer control to "foundit".
    017     1   047          tsearch foundit
                000002
```

Assembler Processing: If a text string is found after the instruction mnemonic, it is assembled as a group after the opcode. The assembler then scans for a symbol which is assumed to be a label. The label is assembled as two bytes, the low order 8 bits of the absolute address is assembled in the first byte. If no text string is found immediately after the mnemonic, the opcode used is 046, and the assembler scans for a label identifier.

TBCNVD, TBCNVO -- convert value on top of stack to a decimal/octal character representation and append to text buffer
160, 156

Description: TBCNVD (160) the value on top of the stack is popped and converted to a decimal character string representation which is appended to the text buffer. Any part of the string which exceeds the maximum text buffer length is chopped. "p1" is unchanged. If "p2" is equal to the number of characters in the buffer, then "p2" is set to the new length. "ln" is set to the new total numer of characters in the buffer.
sp := sp - 1

TBCNVO (156) same as TBCNVD except that the value on top of the stack is converted to an octal representation. The octal representation has the zero character prepended to the number.
sp := sp - 1

Examples:

The expansion of text strings in the listing has been abbreviated in the following example:

```
0       0     140           tbinit
01      0     034           tbappt  "decimal value = "
              020
              144
                    /* TEXT BUFFER: decimal value =
                    /*                  ↑              ↑
                    /*                 p1             p2
                    /* p1 = 0     p2 = ln = 16

023     1     007           loadi   03777   /* load octal 03777
              003777
026     0     160           tbcnvd          /* convert to decimal
                    /* TEXT BUFFER: decimal value = 2047
                    /*                  ↑                  ↑
                    /*                 p1                 p2
                    /* p1 = 0     p2 = ln = 20

027     0     140           tbinit
030     0     034           tbappt  "decimal value = "
              020
              144
052     0     044           tbmp2   -2      /* move p2 left 2
              376
                    /* TEXT BUFFER: decimal value =
                    /*                  ↑              ↑
                    /*                 p1             p2
                    /* p1 = 0     p2 = 14     ln = 16

054     1     006           loadi   0177770 /* load octal for -8
              370
056     0     160           tbcnvd
                    /* TEXT BUFFER: decimal value = -8
                    /*                  ↑                ↑
                    /*                 p1             p2
                    /* p1 = 0     p2 = 14     ln = 18

057     0     140           tbinit
060     0     034           tbappt  "octal value = "
              016
              157
                    /* TEXT BUFFER: octal value =
                    /*                  ↑              ↑
                    /*                 p1             p2
                    /* p1 = 0     p2 = ln = 14

0100    1     006           loadi   63      /* load decimal 63
              077
0102    0     156           tbcnvo          /* append char repres
```



```
                    /* TEXT BUFFER: octal value = 077
                    /*                  ↑                ↑
                    /*                 p1               p2
                    /* p1 = 0     p2 = ln = 17
```

Assembler Processing: None

TBINIT -- initialize text buffer to a null string

140

Description: the text buffer pointers, "p1" and "p2", and the number of characters in the text buffer variable "ln" are set to zero.

sp := sp

Examples:

```
0       0     140           tbinit
                    /* TEXT BUFFER:
                    /*                  ↑
                    /*                p1,p2
                    /* p1 = p2 = ln = 0
```

Assembler Processing: None

TBMP1 -- move text buffer pointer p1

042 or 043

Description: (042) this instruction moves the pointer "p1" within the text buffer. The byte following the opcode contains a two's complement number. If the sign of the number is negative, the pointer is moved to the left and if positive to the right. If the absolute value is:

1-122	p1 is moved this number of character positions (left if negative and right if positive). A user error results if an attempt is made to move p1 before the start (< 0) or to a positon greater than the length (the number of characters in the text buffer)
123	p1 is moved to the next blank character
124	p1 is moved to the next non blank character

```
125     p1 is moved to the next alpha-numeric character
126     p1 is moved to the next non alpha-numeric character
127     p1 is moved to start or end of buffer
128     p1 is moved to other pointer
```

In the above cases in which the pointer is moved to the next blank, non blank, alpha-numeric or non alpha-numeric, if no characters satisfy the criteria in the segment extending in the proper direction, the pointer will not be moved. For example, if the text buffer contains the following:

```
THIS IS TODAY, JANUARY 15, 1977
   ↑                          ↑
  p1                         p2
p1 = 4       p2 = ln = 31
```

and the following statements are applied to the text buffer as is, then:

```
TBMP1  3       /* sets p1 to 7  (blank after 'IS')

TBMP1  -4      /* sets p1 to 0  (1st char in buffer)

TBMP1  123     /* sets p1 to 7  (blank after 'IS')

TBMP1  -123    /* move left to next blank -- there are
               /*  none, so p1 is unchanged

TBMP1  -124    /* move left to next non blank
               /*  which would set p1 = 3 ('S')

TBMP1  127     /* move to end of buffer, p1 = 31
```

sp := sp

(043) "p1" is moved by the amount popped off of the stack. If number is negative, "p1" is moved the absolute value of this number to the left. If the number is positve, the pointer is moved the value to the right. A user error results if an attempt is made to move "p1" before the start of the text buffer (< 0) or to a position greater than the length.
sp := sp - 1

Examples:

```
0     0     042          tbmp1   1       /* move p1 right 1 pos.
            001
02    0     042          tbmp1   -1      /* move p1 left 1 pos.
            377
                  /* an example of moving p1 distance on stack:
04    1     006          loadi   -4      /* load -4 on stack
            374
06    0     043          tbmp1           /* move p1 4 left
```


Assembler Processing: If a number follows the instruction mnemonic, it is assembled as a byte following the opcode. Otherwise the opcode used is 043.

TBMP2 -- move text buffer pointer p2
044 or 045

Description: (044) this instruction moves the pointer "p2" within the text buffer. Identical to TBMP1 except that "p2" is used in place of "p1"
sp := sp

(045) "p2" is moved by the amount popped off of the stack. If the number is negative, "p2" is moved the absolute value of the number to the left. If the number is positive, the pointer is moved the value to the right. A user error will result if an attempt is made to move "p2" before the start of the text buffer (< 0) or to a position greater than the length. SEE TBMP1.
sp := sp - 1

Examples:

```
0     0     044          tbmp2   1       /* move p2 right 1 pos.
            001
02    0     044          tbmp2   -1      /* move p2 left 1 pos.
            377
                  /* an example of moving p2 distance on stack:
04    1     006          loadi   -4      /* load -4 on stack
            374
06    0     045          tbmp2           /* move p2 4 left
07    0     044          tbmp2   128     /* move p2 to p1
            200
```

Assembler Processing: If a number follows the instruction mnemonic, it is assembled as a byte following the opcode. Otherwise the opcode used is 045.

TBLDP1, TBLDP2, TBLDLN -- load value of "p1", "p2", or "ln" on the stack
162, 164, 166

Description: TBLDP1 (162) the value of "p1" is loaded on the stack
sp := sp + 1

TBLDP2 (164) the value of "p2" is loaded on the stack
sp := sp + 1

TBLDLN (166) the value of "ln" which contains the number of characters in the text buffer is loaded on the stack
sp := sp + 1

Examples:

```
0     0    140          tbinit
01    0    034          tbappt  "sample text"
           013
           163
                  /* TEXT BUFFER: sample text
                  /*              ↑          ↑
                  /*             p1         p2
                  /* p1 = 0     p2 = ln = 11

016   1    162          tbldp1
                  /* top of stack contains  0
017   2    164          tbldp2
                  /* top of stack contains 11
020   3    166          tbldln
                  /* top of stack contains 11
```

Assembler Processing: None

TBSTP1, TBSTP2 -- the value on top of the stack is loaded into "p1"/"p2"
170, 172

Description: TBSTP1 (170) the value on top of the stack is popped and loaded into "p1" (one of the text buffer pointers). A user error results if the value is less than zero or greater than the number of characters in the buffer.
sp := sp - 1

TBSTP2 (172) same as TBSTP1 except "p2" is used inplace of "p1"
sp := sp - 1

Examples:


```
0     0    140          tbinit
01    0    034          tbappt  "sample text"
           013
           163
                  /* TEXT BUFFER: sample text
                  /*              ↑          ↑
                  /*             p1         p2
                  /* p1 = 0     p2 = ln = 11

016   1    006          loadi   3
           003
020   0    170          tbstp1          /* move p1 to 3
                  /* TEXT BUFFER: sample text
                  /*                 ↑       ↑
                  /*                p1      p2
                  /* p1 = 3     p2 = ln = 11
```

Assembler Processing: None

5. File Management Instructions:

fload	load file with given name into current active file
fsave	save active file under given name
fdel	delete file with given name
setfn	make active file current
setbn	make block current
seten	make entry current
settln	make text line current
ldfn	load current active file number
ldbn	load current block number
lden	load current entry number
ldtln	load current text line number
ldlsten	load last entry number
ldlsttn	load last text line number
initf	initialize active file to empty
initb	initialize block
delb	delete current block
appe	append entry to current block
inse	insert an entry before current entry
dele	delete current entry
apptl	append text line to current entry
instl	insert text line before current text line
chngl	change current text line
dell	delete current text line
setdy	set dy displacement of block
setdx	set dx displacement of block
lddy	load dy displacement of block
lddx	load dx displacemente of block
setxyt	set xy coordinates of text in entry
ldxyt	load xy coordinates of text in entry
stq	store xy quad word in current entry
stw	store information word in current entry
ldq	load xy quad word from current entry
ldw	load information word from current entry

FLOAD -- copy the file whose name is in the text buffer into the current active file

320

Description: the file whose name is in the text buffer between "p1" and "p2" is loaded into the current active file. If the file is not found, control transfers to the absolute address given in the two bytes following the opcode. Standard UNIX pathnames are allowed. A user error results if "p1" is greater than "p2".
sp := sp

Examples:

```
                    /* Suppose the text buffer contains:
                    /*
                    /* TEXT BUFFER: file: /mnt/pete/dir
                    /*                    ↑           ↑
                    /*                   p1          p2
                    /* p1 = 6     p2 = ln = 19
                    /*
                    /* The following code will load the file
                    /* "/mnt/pete/dir" into active file 0.  If
                    /* the file is not found, control passes
                    /* to "notfound".

    0     1     006          loadi   0
                000
    02    0     176          setfn           /* make active file
                                               /*  0 the current
                                               /*  active file
    03    0     321          fload   notfound
                000000

                    /* The assembler will automatically generate
                    /*  the address 177777 if no label is listed.
                    /*  (a branch to this address in an error)

warning--label assumed to be 177777
    06    0     321          fload           /* no label
                177777

                notfound:
```

Assembler Processing: If no identifier is found after the "FLOAD" mnemonic, a warning message is placed in the listing and the label address 177777 is used.

FSAVE -- save the current active file into the file named in the text buffer.
322

Description: the current active file is copied into the file whose name is in the text buffer between "p1" and "p2". If a file already exists with the name, the old file is deleted. A user error results of "p1" is greater than "p2". Standard UNIX pathnames are valid.
sp := sp 322

Examples:

```
                    /* Suppose the text buffer contains:
                    /*
                    /* TEXT BUFFER: file: /mnt/pete/dir
                    /*                    ↑          ↑
                    /*          .         p1         p2
                    /* p1 = 6     p2 = ln = 19
                    /*
                    /* The following code copies active file
                    /*  zero into the file named:
                    /*  "/mnt/pete/dir".

    0     1    006          loadi   0      /* load 0 on stack
               000
    02    0    176          setfn          /* make active file
                                           /*  zero the current
                                           /*  active file
    03    0    322          fsave          /* save active file 0
```

Assembler Processing: None

FDEL -- delete the file whose name is in the text buffer
324

Description: the file whose name is in the text buffer between "p1" and "p2" is deleted.
sp := sp

Examples:
See FSAVE

Assembler Processing: None

SETFN -- set the current active file number
176

Description: the value on top of the stack is popped and loaded into "curafnum" in the console vector making it the current active file. Active file numbers vary from zero to seven.
sp := sp - 1

Examples:

```
    0     1    006          loadi   0      /* load 0 on stack
               000
    02    0    176          setfn          /* make active file 0
                                           /*  the current active
                                           /*  file.
    03    1    006          loadi   8      /* load constant 8
               010
    05    0    176          setfn          /* make active file 8
                                           /*  the current active
                                           /*  file.
```

Assembler Processing: None

SETBN -- set current block number to value on top of stack
204

Description: the value on top of the stack is popped and stored in the current block number of the current active file head. Until the active file is changed or the current block number is changed, all future references to blocks and sub elements will apply to the block with this number. No verification of the existence of a block with this number as an identifier is made at this time. An error will occur at the time an attempt is made to reference a location in a non-existing block.
sp := sp - 1

Examples:

```
    0     1    006          loadi   1      /* load 1 on stack
               001
    02    0    204          setbn          /* make block 1 the
                                           /*  current block in
                                           /*  the current active
                                           /*  file
```

Assembler Processing: None

SETEN -- set the current entry number in the current block
224

Description: the value on top of the stack is popped and loaded into the "current entry number" in the current block. The entry number corresponding to a particular entry is its position in the block. The first entry in a block is entry number one. If an entry is inserted or deleted, the following entries change number by plus one or minus one respectively. No verification of the existence of an entry having the new number is made during the SETEN instruction. The current text line number in the block head refers to the current text line in the current entry. This value is zeroed in this instruction and in all other instructions which change the current entry.
sp := sp - 1

Examples:

```
0     1     006          loadi   2      /* load 2 on stack
            002
02    0     224          seten          /* set current entry
                                        /*  number in current
                                        /*  block to 2
```

Assembler Processing: None

SETTLN -- set current text line number
240

Description: the value on top of the stack is popped and loaded into the current text line number in the head of the current block. Just as with entries, the number associated with a given text line is its position in the entry. The first line in the entry is number one. No verification of the existence of a text line having the new number is made during the SETTLN instruction.
sp := sp - 1

Examples:

```
0     1     006          loadi   3      /* load 3 on stack
            003
02    0     240          settln         /* make the 3rd text
                                        /*  line in the current
                                        /*  entry the current
                                        /*  text line
```


Assembler Processing: None

LDFN -- load current active file number on stack
200

Description: the current active file number (found in "curafnum" in the console vector) is pushed onto the stack.
sp := sp + 1

Examples:

```
0     1     200          ldfn    /* load current active file #
```

Assembler Processing: None

LDBN -- load current block number on stack
206

Description: the current block number located in the current active file head is loaded onto the stack.
sp := sp + 1

Examples:

```
0     1     206          ldbn    /* load current block #
                                 /* on stack.
```

Assembler Processing: None

LDEN -- load current entry number on stack
226

Description: the value in the current entry number in the current block head is loaded on the stack.
sp := sp + 1

Examples:
(See SETEN)

Assembler Processing: None

LDTLN -- load current text line number on stack
242

Description: the current text line number located in the current block head is loaded on the stack.
sp := sp + 1

Examples:
See example in LDLSTTN

Assembler Processing: None

LDLSTEN -- load the number of the last entry of the current block
230

Description: the block is scanned counting each entry. The total number of entries in the block is loaded on the stack. (the value in the current entry number is not affected)
sp := sp + 1

Examples:
Suppose the following file is the current active file. The LDLSTEN instruction will load a 3 on the stack:

```
/-------------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 034
/      file length=15
/
/      disp    contents
/      0       6       current block number
/      1       13          disp to end of block
/      2       6           block number
/      3       2           current entry number
/      4       0           current text line number
/      5       1           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       1               disp to end of entry
/      10      10              info word 1
/      11      1               disp to end of entry
/      12      20              info word 1
/      13      1               disp to end of entry
/      14      30              info word 1
/
/-------------------------------------------------

035    1     230          ldlsten /* load last entry number (3)
```

Assembler Processing: None

LDLSTTN -- load last text line number in current entry
244

Description: the current entry is scanned beginning to end counting the number of text lines. The total number of lines in the entry is loaded on the stack. The current text line number is not affected.
sp := sp + 1

Examples:
Suppose the following file is the current active file:

```
/-------------------------------------------------
/
/    The following file contains 1 block with
/    3 entries.  The second entry, which is
/    current, contains 3 text lines:
/       "text1"
/       "text2"
/       "text3"
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 071
/      file length=26
/
/      disp    contents
/      0       6       current block number
/      1       24          disp to end of block
/      2       6           block number
/      3       2           current entry number
```

```
/      4       2           current text line number
/      5       1           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       1               disp to end of entry
/      10      10              info word 1
/      11      12              disp to end of entry
/      12      20              info word 1
/      13      0               y coord of text
/      14      0               x coord of text
/      15      5 164           char cnt, 't'
/      16      145 170         'ex'
/      17      164 061         't1'
/      18      5 164           char cnt, 't'
/      19      145 170         'ex'
/      20      164 062         't2'
/      21      5 164           char cnt, 't'
/      22      145 170         'ex'
/      23      164 063         't3'
/      24      1               disp to end of entry
/      25      30              info word 1
/
/-----------------------------------------------

072    1    244          ldlsttn          /* load last text
                                          /* line number
                  /* top of stack now contains 3.  Notice that
                  /*  the current text line number is unaffected.
073    2    242          ldtln            /* load current text
                                          /*  line number
                  /* top of stack now contains 2.  Change to
                  /*  third entry which does not have any
                  /*  text lines.
074    3    006          loadi   3
            003
076    2    224          seten            /* make 3rd entry current
077    3    244          ldlsttn          /* load last text line
                                          /*  number
                  /* top of stack now contains a zero.
```

Assembler Processing: None

INITF -- initialize active file to empty
202

Description: the value on top of the stack is popped and the corresponding active file is made current. The file is initialized to the length of the file head and the current block number in the file head is set to zero.
sp := sp - 1

Examples:

```
0      1     006          loadi   0       /*load 0 on stack
             000
02     0     202          initf           /* initialize file 0
```

Assembler Processing: None

INITB -- initialize block in current active file
210

Description: a new empty block is initialized at the end of the current active file. The initialization process consists of constructing a block header containing information about the new block. A standard block head has the following form:

- word 0 the displacement to the end of the block. Total block length is the contents of word 0 plus 1.
- word 1 block id number. (Each block in a file should have a unique identifying number)
- word 2 current entry number.
- word 3 current text line number.
- word 4 number of information words per entry. (Each entry within a block contains the same number of information words)
- word 5 number of xy quads per entry. (Each entry within a block contains the same number xy quads)
- word 6 dy, the displacement in the y direction given to each entry in the block.
- word 7 dx, the displacement in the x direction given to each entry in the block.

The top three levels of the stack are popped and used to initialize the header. The top level contains the number of xy quads per entry. The second level contains the number of information words per entry and finally, the third level contains the id number of the new block. The displacement to the end of the block (word 0) is initialized to 7. The current entry number, current text line number, dy, and dx are set to zero. The

current block number in the file head is set to the id number of the new block making it the current block.
sp := sp - 3

Examples:

```
                        /* The following program initializes active file
                        /*  0 then initializes block 6 which will contain
                        /*  3 information words and 2 xy quads per entry.
                        /*  The file is dumped using the debugging aid
                        /*  DUMPAF.
0       1       006             loadi   0       /* load 0 on stack
                000
02      0       202             initf           /* initialize file
03      1       006             loadi   6       /* block #
                006
05      2       006             loadi   3       /* # of info words/entry
                003
07      3       006             loadi   2       /* # of xy quads/entry
                002
011     0       210             initb           /* initialize block 6
012     0       104             dumpaf          /* dump file
```

```
/-----------------------------------------------
/
/    The following listing was produced by the DUMPAF
/    instruction:
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 012
/      file length=9
/
/      disp    contents
/      0       6       current block number
/      1       7           disp to end of block
/      2       6           block number
/      3       0           current entry number
/      4       0           current text line number
/      5       3           number of info words/entry
/      6       2           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/
/-----------------------------------------------
```

Assembler Processing: None

DELB -- delete current block
212

Description: the current block in the current active file is deleted and the current block number in the file head is set to zero.
sp := sp

Examples:
Suppose the current active file is as follows:

```
/-----------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 007
/      file length=30
/
/      disp    contents
/      0       6       current block number
/      1       11          disp to end of block
/      2       6           block number
/      3       2           current entry number
/      4       0           current text line number
/      5       1           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       1               disp to end of entry
/      10      10              info word 1
/      11      1               disp to end of entry
/      12      20              info word 1
/      13      16          disp to end of block
/      14      5           block number
/      15      1           current entry number
/      16      1           current text line number
/      17      0           number of info words/entry
/      18      0           number of xy quads/entry
/      19      0           dy
/      20      0           dx
/      21      8               disp to end of entry
/      22      0               y coord of text
/      23      0               x coord of text
/      24      11 163          char cnt, 's'
/      25      141 155         'am'
/      26      160 154         'pl'
/      27      145 040         'e '
/      28      164 145         'te'
/      29      170 164         'xt'
/
/-----------------------------------------------
```

The following instruction deletes the current block (block 6):

```
010     0       212             delb
```

The active file now appears as:

```
/-----------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 011
/      file length=18
/
/      disp    contents
/      0       0       current block number
/      1       16          disp to end of block
/      2       5           block number
/      3       1           current entry number
/      4       1           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       8               disp to end of entry
/      10      0               y coord of text
/      11      0               x coord of text
/      12      11 163          char cnt, 's'
/      13      141 155         'am'
/      14      160 154         'pl'
/      15      145 040         'e '
/      16      164 145         'te'
/      17      170 164         'xt'
/
/-----------------------------------------------
```

Assembler Processing: None

APPE -- append entry to current block
232

Description: an entry is created at the end of the current block with the number of information words and xy quad words as indicated in the block head. The length of a newly created entry is the number of information words plus four times the number of xy quads plus one. The current entry number in the block head is set to to the number of the new entry.
sp := sp

Examples:
Suppose the following file is the current active file:


```
/-----------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 012
/      file length=9
/
/      disp    contents
/      0       6       current block number
/      1       7           disp to end of block
/      2       6           block number
/      3       0           current entry number
/      4       0           current text line number
/      5       2           number of info words/entry
/      6       1           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/
/-----------------------------------------------

013    0     232           appe    /* append an entry
014    0     104           dumpaf  /* dump active file

/-----------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 014
/      file length=16
/
/      disp    contents
/      0       6       current block number
/      1       14          disp to end of block
/      2       6           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       2           number of info words/entry
/      6       1           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       6               disp to end of entry
/      10      0               info word 1
/      11      0               info word 2
/      12      0               y1 of quad 1
/      13      0               x1
/      14      0               y2
/      15      0               x2
/
/-----------------------------------------------
```

Assembler Processing: None

INSE -- insert a new entry before current entry in block
234

Description: an entry is created before the current entry with the number of information words and xy quads specified in the block header. The amount of space inserted is equal to the number of information words plus four times the number of xy quads plus one. The current entry number in the block head is unchanged.
sp := sp

Examples:

```
013    0     104          dumpaf  /* dump active file

/-----------------------------------------------
/
/   The above DUMPAF produced the following
/   listing:
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 013
/      file length=16
/
/      disp    contents
/      0       6       current block number
/      1       14          disp to end of block
/      2       6           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       2           number of info words/entry
/      6       1           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       6               disp to end of entry
/      10      10              info word 1
/      11      20              info word 2
/      12      100             y1 of quad 1
/      13      200             x1
/      14      300             y2
/      15      400             x2
/
/-----------------------------------------------

014    0     234          inse    /* insert an entry
015    0     104          dumpaf  /* dump active file

/-----------------------------------------------
/
/    The active file now has the following form:
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 015
/      file length=23
/
/      disp    contents
/      0       6       current block number
/      1       21          disp to end of block
/      2       6           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       2           number of info words/entry
/      6       1           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       6               disp to end of entry
/      10      0               info word 1
/      11      0               info word 2
/      12      0               y1 of quad 1
/      13      0               x1
/      14      0               y2
/      15      0               x2
/      16      6               disp to end of entry
/      17      10              info word 1
/      18      20              info word 2
/      19      100             y1 of quad 1
/      20      200             x1
/      21      300             y2
/      22      400             x2
/
/-----------------------------------------------
```

Assembler Processing: None

DELE -- delete current entry
236

Description: the current entry in the current block is deleted. The remainder of the file is moved up and the displacements to the end of the block and file are adjusted. If the deleted entry was the last in the block, the current entry number is zeroed, otherwise the number is not affected. The current text line number in the head of the current block is zeroed.
sp := sp

Examples:

```
9      0     236          dele    /* delete current entry
```

Assembler Processing: None

APPTL -- append text line in text buffer to current entry
246

Description: the text in the text buffer between p1 and p2 is appended to the current entry. The current text line number in the block head is set to the number of the new text line. A user error results if "p2" less than "p1". If this is the first text line in the entry, space is created before the text line for the coordinates of the text (2 words). The text in the entry is packed one character to a byte and is preceded by a byte containing the character count of the text. The file is word addressable and therefore a string with an even number of characters is padded with a zero byte so that the character count and text string ends on a word boundary.
sp := sp

Examples:
Suppose we have just created a block and entry.

```
013    0      104          dumpaf  /* dump the current
                                   /*  active file

/-----------------------------------------------
/
/    listing produced by DUMPAF:
/
/      ACTIVE FILE 0  'GPIL/ACTFILA'    pc = 013
/      file length=10
/
/      disp    contents
/      0       1       current block number
/      1       8           disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       0               disp to end of entry
/
/-----------------------------------------------

014    0    034          tbappt  "sample text"
            013
            163
                  /* TEXT BUFFER: sample text
                  /*              ↑          ↑
                  /*             p1          p2
                  /* p1 = 0     p2 = ln = 11
                  /*
                  /* append text in buffer between p1 and p2
```



```
                  /*  ("sample text") to current entry:
031    0    246          apptl

032    0    042          tbmp1   5       /* move p1 right 4
            005
034    0    044          tbmp2   -2      /* move p2 left 2
            377
                  /* TEXT BUFFER: sample text
                  /*                   ↑   ↑
                  /*                  p1   p2
                  /* p1 = 5     p2 = 9     ln = 11
                  /*
                  /* append text in buffer between p1 and p2
                  /*  ("e te") to current entry
036    0    246          apptl
037    0    104          dumpaf

/-----------------------------------------------
/
/    listing produced by DUMPAF:
/
/      ACTIVE FILE 0  'GPIL/ACTFILA'    pc = 037
/      file length=21
/
/      disp    contents
/      0       1       current block number
/      1       19          disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       2           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       11              disp to end of entry
/      10      0               y coord of text
/      11      0               x coord of text
/      12      11 163          char cnt, 's'
/      13      141 155         'am'
/      14      160 154         'pl'
/      15      145 040         'e '
/      16      164 145         'te'
/      17      170 164         'xt'
/      18      5 145           char cnt, 'e'
/      19      040 164         ' t'
/      20      145 0           'e'
/
/-----------------------------------------------
```

Assembler Processing: None

INSTL -- insert text line before current text line
250

Description: the text in the text buffer between p1 and p2 is inserted before the current text line in the current entry. The current text line number in the block head is unchanged.
sp := sp

Examples:
See APPTL

Assembler Processing: None

CHNGL -- change current text line
254

Description: the current text line in the current entry is replaced by the text in the text buffer between p1 and p2. Space is inserted or deleted in the file if the new line occupies more or less space than the old text line. The current text line pointer in the block head is not changed.
sp := sp

Examples:
Suppose the following file is the current active file:

```
/-------------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 071
/      file length=20
/
/      disp    contents
/      0       1       current block number
/      1       13          disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       1           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       10              disp to end of entry
/      10      200             y coord of text
```



```
/      11      100             x coord of text
/      12      6 154           char cnt, '1'
/      13      151 156         'in'
/      14      145 040         'e '
/      15      061             '1'
/      16      6 154           char cnt, '1'
/      17      151 156         'in'
/      18      145 040         'e '
/      19      062             '2'
/
/-------------------------------------------------

                    /* Suppose the text buffer is as
                    /*  follows:
                    /*
                    /* TEXT BUFFER: sample text
                    /*                ↑    ↑
                    /*               p1   p2
                    /* p1 = 1     p2 = 6     ln = 11

072     0     254           chngl
```

The following listing is the active file after the CHNGL instruction. Notice that the character string between p1 and p2 is "ample" and that the new text group is 1 word shorter than the previous.

```
/-------------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 073
/      file length=19
/
/      disp    contents
/      0       1       current block number
/      1       17          disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       1           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       9               disp to end of entry
/      10      200             y coord of text
/      11      100             x coord of text
/      12      5 141           char cnt, 'a'
/      13      155 160         'mp'
/      14      154 145         'le'
/      15      6 154           char cnt, 'l'
/      16      151 156         'in'
/      17      145 040         'e '
/      18      062             '2'
/
/-------------------------------------------------
```

Assembler Processing: None

DELL -- deleted current text line
252

Description: the current text line in the current entry is deleted. The remaining text lines are shifted up. If the line deleted was the last line in the entry, the space provided for the text coordinates is also deleted.
sp := sp

Examples:

```
047    0     104          dumpaf  /* dump current active file

/-------------------------------------------------
/
/    listing produced by DUMPAF:
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 047
/      file length=20
/
/      disp    contents
/      0       1       current block number
/      1       18          disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       1           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       10              disp to end of entry
/      10      200             y coord of text
/      11      100             x coord of text
/      12      6 154           char cnt, '1'
/      13      151 156         'in'
/      14      145 040         'e '
/      15      061             '1'
/      16      6 154           char cnt, '1'
/      17      151 156         'in'
/      18      145 040         'e '
/      19      062             '2'
/
/-------------------------------------------------

050    0     252          dell    /* delete current text line
                                   /* (line 1 of current entry)
```



```
051    0     252          dell    /* delete current text line
                                   /* (line 1 of current entry)
052    0     104          dumpaf  /* dump current active file

/-------------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 052
/      file length=10
/
/      disp    contents
/      0       1       current block number
/      1       8           disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       0           number of info words/entry
/      6       0           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       0               disp to end of entry
/
/-------------------------------------------------
```

Assembler Processing: None

SETDY, SETDX -- pop top of stack into dy or dx in current block
220, 222

Description: SETDY (220) the value on top of the stack is popped and loaded into "dy" in the block head of the current active file.
"dy" and "dx" are 16 bit values in the block head which are added to each of the x y pairs in the block. These values are used to translate the pictorial information in the block to different screen coordinates.
sp := sp -1

SETDX (222) same as SETDY except the value is stored in "dx".
sp := sp - 1

Examples:
See LDDY

Assembler Processing: None

LDDY, LDDX -- load dy, dx from current block
214, 216

Description: LDDY (214), the contents of "dy" in the current block head of the current active file is loaded onto the stack.
"dy" and "dx" are 16 bit values in the block head which are added to each of the x y pairs in the block. These values are used to translate the pictorial information in the block to different parts of the screen.
sp := sp + 1

LDDY (216), same as LDDY except "dx" is loaded onto stack
sp := sp + 1

Examples:
Suppose the following file is the current active file. The current block is block 6.

```
/-----------------------------------------------
/      ACTIVE FILE 0  'GIRL/ACTFILA'    pc = 051
/      file length=13
/
/      disp    contents
/      0       6       current block number
/      1       11          disp to end of block
/      2       6           block number
/      3       2           current entry number
/      4       0           current text line number
/      5       1           number of info words/entry
/      6       0           number of xy quads/entry
/      7       140         dy
/      8       -140        dx
/      9       1               disp to end of entry
/      10      10              info word 1
/      11      1               disp to end of entry
/      12      20              info word 1
/
/-----------------------------------------------
```

The following code will load a 140 and a -140 on the stack:

```
0      1     214          lddy            /* load dy (= 140)
01     2     216          lddx            /* load dx (=-140)
```

Assembler Processing: None

SETXYT -- set display coordinates of text in current entry
260

Description: the top two values of the stack are popped and stored in the x y text coordinates in the current entry. The y coordinate is the value taken from the top level of the stack. A user error results if there are no text lines in the current entry.
sp := sp - 2

Examples:
See APPTL

Assembler Processing: None

LDXYT -- load text coordinates of current entry
256

Description: the x y text coordinates in the current entry are loaded onto the stack. The y coordinate is on top. An error occurs if one attempts to load the text coordinates and there are none in the current entry. Text coordinates do not exist in the entry if there are no text lines.
sp := sp + 2

Examples:
See APPTL

Assembler Processing: None

STQ -- store xy quad word in current entry
122 or 123

Description: (122) the byte after the opcode contains an 8 bit number which is assumed to be positive (ie 0-255). The top of the stack is popped and the value is stored in the xy quad word having the index of the 8 bit positive number. The index of the first coordinate is one.
sp := sp - 1

(123) the index to the particular xy quad coordinate is popped off of the stack. The next level is popped off the stack and stored in the indexed xy quad word. Any attempt to store a coordinate outside of the xy quad area in the current entry results in a user error.
sp := sp - 2

Examples:
The following program constructs a file with 2 xy quads (and 1 information word) per entry.

```
0     1    006          loadi   0
           000
02    0    202          initf
03    1    006          loadi   1       /* block number
           001
05    2    006          loadi   1       /* # of infor words/entry
           001
07    3    006          loadi   2       /* # of xy quads/entry
           002
011   0    210          inith           /* create block 1
012   0    232          appe            /* append an entry

013   1    007          loadi   600     /* x2
           001130
016   2    007          loadi   500     /* y2
           000764
021   3    007          loadi   400     /* x1
           000620
024   4    007          loadi   300     /* y1 of quad 1
           000454
027   3    122          stq     1       /* store 300 into 1st
           001
                                        /*  xy quad word
031   2    122          stq     2       /* store 400 into 2nd
           002
                                        /*  xy quad word
033   1    122          stq     3       /* etc.
           003
035   0    122          stq     4
           004

037   1    007          loadi   775     /* will be x2 of quad 2
           001407
042   2    007          loadi   675     /* will be y2 of quad 2
           001243
045   3    007          loadi   575     /* will be x1 of quad 2
           001077
050   4    007          loadi   475     /* will be y1 of quad 2
           000733
053   3    122          stq     5       /* store y1 of quad 2
           005
055   4    006          loadi   6       /* alternate method
           006
```



```
057   2    123          stq             /* store x1 of quad 2
060   3    006          loadi   7       /* store y2 of quad 2
           007
062   1    123          stq
063   0    122          stq     8       /* store x2 of quad 2
           010
065   0    104          dumpaf          /* dump active file

/-----------------------------------------------
/
/    The following listing was produced by the DUMPAF
/    command:
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 065
/      file length=19
/
/      disp    contents
/      0       1       current block number
/      1       17          disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       1           number of info words/entry
/      6       2           number of xy quads/entry
/      7       0           dy
/      8       0           dx
/      9       9               disp to end of entry
/      10      0               info word 1
/      11      300             y1 of quad 1
/      12      400             x1
/      13      500             y2
/      14      600             x2
/      15      475             y1 of quad 2
/      16      575             x1
/      17      675             y2
/      18      775             x2
/
/-----------------------------------------------

                        /* the following STQ would at execution
                        /*  time result in a user error.  Note
                        /*  however that the assembler generates
                        /*  a LOADI if the displacement to the
                        /*  particular coordinate is beyond
                        /*  255.
066   1    006              loadi   0       /* coordinate value
           000
070   1    007'
           000404
073   0    123              stq     263     /* store x2 of quad 65
```

Assembler Processing: If a no number is found after the instruction mnemonic then the opcode 123 is used. If a number is found and it is between 0 and 255 it is assembled into the byte following

the opcode. If the number is greater than 255, a LOADI instruction is generated to load the number on the stack and then the opcode 123 is used for the STO.

STW -- store information word in current entry
124 or 125

Description: (124) the byte after the opcode contains an 8 bit number which is taken to be positive (ie 0-255). The top of the stack is popped and stored in the information word of the current entry having the 8 bit number as an index. The index of the first information word is 1. Any attempt to reference an information word outside the space set aside at the creation of the entry results in a user error.
sp := sp - 1

(125) the index to the particular information word is popped off of the stack. The next level is then popped and stored in the indexed information word of the current entry. Any attempt made to reference an information word beyond the created space reults in a user error.
sp := sp - 2

Examples:
The following program creates a file with 3 information words (and 1 xy quad) per entry.

```
0      1    006          loadi   0
            000
02     0    202          initf
03     1    006          loadi   1       /* block number
            001
05     2    006          loadi   3       /* # of infor words/entry
            003
07     3    006          loadi   1       /* # of xy quads/entry
            001
011    0    210          initb           /* create block 1
012    0    232          appe            /* append an entry

013    1    006          loadi   10      /* info for word 1
            012
015    2    006          loadi   20      /* info for word 2
            024
017    1    124          stw     2       /* store info word 2
            002
021    0    124          stw     1       /* store info word 1
            001
```



```
023    1    006          loadi   30      /* info for word 3
            036
025    2    006          loadi   3       /* index to info word
            003
027    0    125          stw             /* store info word 3
030    0    104          dumpaf          /* dump file

/-----------------------------------------------
/
/   The following listing was produced by the DUMPAF
/   instruction:
/
/     ACTIVE FILE 0  'GIHL/ACTFILA'    pc = 030
/     file length=17
/
/     disp    contents
/     0       1       current block number
/     1       15          disp to end of block
/     2       1           block number
/     3       1           current entry number
/     4       0           current text line number
/     5       3           number of info words/entry
/     6       1           number of xy quads/entry
/     7       0           dy
/     8       0           dx
/     9       7               disp to end of entry
/     10      10              info word 1
/     11      20              info word 2
/     12      30              info word 3
/     13      0               y1 of quad 1
/     14      0               x1
/     15      0               y2
/     16      0               x2
/
/-----------------------------------------------

                    /* the following STW would at execution
                    /*  time result in a user error due to
                    /*  the fact that there are only 3 infor
                    /*  words per entry.  Note however that
                    /*  the assembler generates a LOADI if
                    /*  the dispacement will not fit in the
                    /*  byte following the opcode.
031    1    006          loadi   -1      /* info word to store
            377
033    1    007'
            000400
036    0    125          stw     256     /* store into iw 256
```

Assembler Processing: See STO

LDQ -- load xy quad word from current entry
126 or 127

Description: (126) the byte after the opcode contains an 8 bit number which is taken to be positive (ie 0-255). The xy quad coordinate which is at this displacement in the xy quad space of the current entry is loaded onto the stack. (The displacement to the first coordinate is 1). Any attempt to reference an xy quad word outside of the space set aside at the creation of the entry results in a user error.
sp := sp + 1

(127) the displacement or index to the particular xy quad word is popped off of the stack. The coordinate at this displacement in the xy quad space of the current entry is loaded onto the stack.
sp := sp

Examples:
See STQ for more examples. Suppose that the xy quad area in the current entry contains the following information:

```
        100     y1 of quad 1
        200     x1 of quad 1
        300     y2 of quad 1
        400     x2 of quad 1
        575     y1 of quad 2
        675     x1 of quad 2
        775     y2 of quad 2
        875     x2 of quad 2

0       1       126             ldq     1       /* load y1 of quad 1
                001
02      2       126             ldq     2       /*      x1         1
                002
                        /* The top of the stack now contains 200,
                        /*  the next level down contains 100.
04      3       006'
                007
06      3       005             load    7       /* get ready to load
                                                /*  the 7th coord
07      3       127             ldq             /* load y2 of quad 2
                        /* The top of the stack now contains 775.
                        /*  The last example demonstrates the
                        /*  assembler automatically generating a
                        /*  LOADI for a displacement which can
                        /*  not fit in the byte after the opcode.
010     5       007'
                000401
013     4       127             ldq     257     /* load y1 of the
                                                /*  65th quad.
```


Assembler Processing: See STQ

LDW -- load information word from current entry
130 or 131

Description: (130) the byte following the opcode contains an 8 bit number which is taken to be positive (ie 0-255). The information word at this index in the current entry is loaded onto the stack. (The index of the first information word is 1). Any attempt to reference an information word outside the space set aside at the creation of the entry results is a user error.
sp := sp + 1

(131) the index to the particular information word is popped off the stack. Then the information word at this index in the current entry is loaded onto the stack.
sp := sp

Examples:
See also STW. Suppose that the information words in the current entry contain:

```
        10      info word 1
        20      info word 2
        30      info word 3

0       1       130             ldw     1       /* load 1st info word
                001
02      2       130             ldw     2       /* load 2nd info word
                002
                        /* The top of the stack now contains 20, the
                        /*  next level down, 10
04      3       006             loadi   3       /* get ready to load 3rd
                003
                                                /*  info word on stack
06      3       131             ldw             /*  get index from stack
                                                /*  and load word
                        /* Top of stack now contains 30.  The fol-
                        /*  lowing example would cause a user if
                        /*  there were less than 256 information
                        /*  words per entry.  However, the example
                        /*  demonstrates the assembler automat-
                        /*  ically generating a LOADI for indexes
                        /*  which can not fit in the byte following
                        /*  the opcode.
07      5       007'
                000400
012     4       131             ldw     256     /* load the 256th iw
```

Assembler Processing: See STQ

6. Display Instructions:

dispf	display current active file
dispb	display current block
dispe	display current entry
displtl	display current text line
dispq	display xy quad
displtxt	display text
erase	erase screen

DISPF, DISPB, DISPE, DISPTL -- display current file, block, entry and text line
300, 302, 304, 310

Description: these instructions display particular structures of the current active file. xy quads are displayed as line segments and the text in entries is displayed a line at a time with each subsequent text line displayed beneath the previous. In all instructions, the displacements "dx" and "dy" stored in each block head are added to the coordinates of line segments and the text.

DISPF (300) the entire active file is displayed on the screen

DISPB (302) the entire current block is displayed on the screen

DISPE (304) the entire current entry is displayed on the screen

DISPTL (310) the current text line is displayed on the screen
sp := sp

Examples:
The following code displays the current text line a total of four times, the current entry a total of three, the current block twice and the entire file once.

```
0     0     310          displtl  /* display current text line
01    0     304          dispe    /* display current entry
02    0     302          dispb    /* display current block
03    0     300          dispf    /* display current file
```

Assembler Processing: None

DISPQ -- display xy quad in current entry
306

Description: the value on top of the stack is popped and used to index into the xy quads in the current entry. The xy quad at this index is displayed on the screen as a line. The displacements "dx" and "dy" in the block head are added to the coordinates. The index of the first xy quad is one.
sp := sp - 1

Examples:
Suppose the following file is the current active file:

```
/-----------------------------------------------
/
/      ACTIVE FILE 0  'GIML/ACTFILA'    pc = 074
/      file length=19
/
/      disp    contents
/      0       1       current block number
/      1       17          disp to end of block
/      2       1           block number
/      3       1           current entry number
/      4       0           current text line number
/      5       1           number of info words/entry
/      6       2           number of xy quads/entry
/      7       10          dy
/      8       -20         dx
/      9       9               disp to end of entry
/      10      10              info word 1
/      11      600             y1 of quad 1
/      12      500             x1
/      13      100             y2
/      14      500             x2
/      15      600             y1 of quad 2
/      16      200             x1
/      17      600             y2
/      18      800             x2
/-----------------------------------------------
```

The following code displays quads one and two in the current entry which form a "T" on the screen. The first quad is the vertical line displayed from point (480,610) to point (480, 110). The second quad is the horizontal line extending from point (180, 610) to point (780, 610).

```
075     1     006           loadi   1
              001
077     0     306           dispq           /* display 1st quad
0100    1     006           loadi   2
              002
0102    0     306           dispq           /* display 2nd quad
```

Assembler Processing: None

DISPTXT -- display text on the screen at coordinates on stack
312 or 313

Description: (312) the text group following the opcode is displayed on the screen at the x and y values popped off the stack. The y coordinate is in the top level. (a text group consists of a byte containing a character count followed by that number of ascii characters packed one to a byte.)
sp := sp - 2

(313) same as above except the text between "p1" and "p2" is used in place of text following the opcode. An error results if "p1" is greater than "p2".
sp := sp - 2

Examples:

```
0       1     006           loadi   100     /* load x coord
              144
02      2     007           loadi   300     /* load y coord
              000454
05      0     312           disptxt "sample text"
              013
              163

                    /* Suppose the text buffer contains:
                    /*
                    /* TEXT BUFFER: abcdefghijkl
                    /*                    *       *
                    /*                   p1      p2
                    /* p1 = 2      p2 = 10      ln = 12
                    /*
                    /* The following code will display
                    /* "cdefghij" 1 line below "sample text"

022     1     006           loadi   100     /* load x coord
              144
024     2     007           loadi   200     /* load y coord
              000430
027     0     313           disptxt
```

Assembler Processing: If text follows the instruction mnemonic, it is assembled as a text group after the opcode. If text does not follow the mnemonic, the opcode used is 313.

ERASE -- erase terminal screen
314

Description: the terminal screen is erased.
sp := sp

Examples:
0 0 314 erase

Assembler Processing: None

7. Procedure Invocation and Return Instructions:

calli	call internal (procedure within current program)
calle	call external (procedure outside of current program)
return	return from a procedure

CALLI -- internal call
114

Description: instruction which invokes a procedure within the current program (ie procedure in the current object file). The number of parameters plus extra levels to pop off when returning are in the first byte after the opcode. The next two bytes contain the 16 bit absolute address of the new procedure. The low order 8 bits of the address are in the first byte.

For each call, a new level is loaded onto the procedure stack which is internal to the user's console vector. Stored in the procedure stack are:

1. the return address
2. the current stack level minus any levels generated prior to the call which are considered part of the call. The number of generated levels is contained in the first byte following the opcode.
3. a pointer to the object file's name which is copied from the outer level.
4. a pointer to the current grid which is also copied from the outer level (ie current grid is inherited).
5. pointer to base of outer procedure level.

sp := sp - n where n is the value of the 1st byte after the opcode

Examples:

This is one of the instructions in which the assembler will automatically generate instructions for the programmer. The assembler generates the necessary loads for the parameter links and any literal constants passed as variables. Each procedure expects the address of all formal parameters to be on the stack upon entering the procedure. GAS scans each call statement and first generates any loads for contants passed as actual parameters and then generates a LOADI for each variable's stack address field. For example:

```
                    /* In the following calls, the address of
                    /*  "crunch" has been edited in.  As with
                    /*  labels, if the procedure lies after the
                    /*  reference in the source, the address
                    /*  of the procedure is not known when
                    /*  the listing is generated.

        0           procedure main()
                           dclglob z

0       1    006           dclloc  x
             000
02      2    006           dclloc  y
             000

                    /* The assembler will generate the necessary
                    /*  LOADI instructions for the addresses of
                    /*  the two parameters x and y.  (generated
                    /*  instructions appear in the listing with
                    /*  a single quote after the opcode)  In the
                    /*  following example x and y are local
                    /*  variables and the correct stack address
                    /*  is the displacement from the variable to
                    /*  the generated level containing the stack
                    /*  address.

04      3    006'
             002
06      4    006'
             002
010     2    114           calli  crunch(x,y)
             002
             000037
                    /* In the next example, the global variable
                    /*  z is passed as parameter.  Notice that
                    /*  the address field to be loaded is the
                    /*  global address of the variable.
014     3    006'
             377
016     4    006'
             003
020     2    114           calli  crunch(z,x)
             002
             000037
                    /* In the next example, a literal constant
                    /*  will be passed as a parameter.  The
                    /*  actual constant must be generated before
                    /*  the parameter address field because the
                    /*  sub procedure expects the address fields
                    /*  of the formal parameters to be continuous
                    /*  and in the correct order at the top of
                    /*  the stack.  Notice that the number of
                    /*  levels generated for this call is 3
                    /*  instead of 2 as in the previous example.
024     3    006'
             017
026     4    006'
             002
030     5    006'
             002
032     2    114           calli  crunch(y, 017)
             003
             000037
036     2    120           return

                    /* Define new procedure "crunch" with formal
                    /*  parameters "formalp1" and "formalp2".
                    /*  The scope of these identifiers is "crunch"
             2      procedure crunch(formalp1, formalp2)
037     3    006'
             377
```

```
041     4       007'
                040003
044     2       114             calli  switch(z, formalpl)
                002
                000051
050     2       120             return
                        /* In the above call to "switch" a global and
                        /*  a formal parameter are passed.  The address
                        /*  field generated for a formal parameter is
                        /*  similar to a local variable except that
                        /*  the indirect bit is set (second most high
                        /*  order bit) indicating that the address
                        /*  field of the parameter lies further down
                        /*  the stack.


        2               procedure switch(a, b)
051     2       120             return
```

Assembler Processing: The assembler scans for an identifier, which is assumed to be that of a procedure, after the CALLI mnemonic. If the identifier is not in the symbol table, an entry is created with the attribute PROC _UDEF. An entry is also entered in the forward reference list so that the address of this procedure can be filled in at the end of pass 1. If the procedure identifier has been defined the address is saved for assembly at the end of the CALLI instruction. Next, the assembler expects to find a left parenthesis. If no parameters are found, the CALLI opcode is assembled followed by zero (no extra stack generations) and finally the absolute address of the procedure as two bytes.

If parameters are found, the load immediate instructions for all constants are generated first followed by the loads for the parameter address fields. The opcode for the CALLI instruction is assembled followed in the next byte by the total number of generated load immediate instructions. The final two bytes of the instruction contains the absolute address of the procedure.

CALLE -- call to external program
116 or 117

Description: (116) this instruction allows for entering GIML programs other than the current one as a procedure invocation. The byte after the opcode contains the number of generated loads preceding the CALLE which are to be removed upon returning to the current program file. The generated loads are for parameter address fields and constants passed as parameters. The name of the new object file follows the second byte of the instruction

in the form of a standard text group (character count followed by ascii characters). The final two bytes contain the absolute address of a program location that is loaded into the program counter if the external file is not found. Execution begins in the new file at the first location.

For each call a new level is loaded onto the procedure stack which is internal to the user's console vector. Stored in the procedure stack at this time are:

1. the return address.
2. the current stack level minus the number of generated levels.
3. the new object file name.
4. a pointer to the new object file name.
5. a pointer to the current grid which is copied from the outer level (ie current grid is inherited).
6. pointer to the base of the outer procedure level.

sp := sp - n where n is the number of generated loads

(117) identical to above except that the new program file name is in the text buffer between "p1" and "p2". A user error will result if "p2" is less than or equal to "p1".

sp := sp - n where n is the number of generated loads (ie value in byte following opcode)

Examples:

This is one of the instructions in which the assembler will automatically generate instructions. The instructions generated in this case are LOADI's for the parameter address fields and any constants used as parameters. The generation of these instructions proceeds exactly as in CALLI and the reader should refer there for futher explanation.

```
        0               procedure main()
                                dclglob z

0       1       006             dclloc  x
                009
02      2       006             dclloc  y       -1
                377

04      2       010     notfound:  move x       /* move y into x
                001

                        /* the following statment calls the program
                        /*  "extern/proc".  If the file is not found
                        /*  control passes to the statement labeled
                        /*  "notfound".

06      3       006'
                002
010     4       006'
                002
```

```
012     5       006'
                377
014     2       116             calle "extern/prog"(x,y,z) notfound
                003
                013
                145
                170
                164
                145
                162
                156
                057
                160
                162
                157
                147
                000004
                        /* Suppose the text buffer contains the
                        /*  following:
                        /*
                        /* TEXT BUFFER: file: games
                        /*                    ↑    ↑
                        /*                   p1    p2
                        /* p1 = 6    p2 = ln = 11
                        /*
                        /* In the following call, which uses the
                        /*  text between p1 and p2 for the name
                        /*  of the file containing the interpreter
                        /*  program, no label has been given for
                        /*  the "not found" branch.  If none is
                        /*  given, GAS uses the address 177777
                        /*  which is the highest addressable
                        /*  program location.  Since at least
                        /*  two bytes are needed to transfer
                        /*  program control, branching to this
                        /*  location is guaranteed to cause a
                        /*  user error.  (In fact addressing
                        /*  a location greater than the program
                        /*  size results in an error.)
034     3       006'
                012
036     4       006'
                001
warning--assumed branch label set at 177777
040     2       117             calle (10)      /* call extern "games"
                002
                177777
```

Assembler Processing: The assembly of this instruction is a little complicated due to the fact that all loads must be generated before the CALLE opcode. The load instructions for any literal constants used as actual parameters are generated first. Then for each actual parameter, a load instruction is generated for the appropriate address field. The opcode for the CALLE instruction is assembled next. If no external program name is present, the low order bit of the opcode is set. The next assebled byte contains the total number of load instructions required for the parameter linkage. This includes the number of load instructions generated for literal constants used as parameters. If an external program name is present, it is assembled at this point as a standard text group. The final two bytes of the instruction contain the absolute address of the label. As mentioned in the example, if no label is found, 0177777 is used for the address and a warning message is printed in the listing file.

RETURN -- return from a procedure
120

Description: return from an external or internal procedure. The procedure stack pointer is moved to the outer procedure level popping the current level. The stack pointer is set to its value prior to the call which had been saved in the return stack, and the program counter is loaded with the return address also saved in the procedure stack. If control is returning to a different interpreter program (ie different file name) the program name pointed to by "name pointer" in the new level becomes the new object file.
sp := sp

Examples:

```
                                /* The assembler directive ALLOCGLOBS is
                                /*  used to create the space at the base
                                /*  of the stack for globals.  The direc-
                                /*  tive generates a PUSH opcode followed
                                /*  by a null byte which is filled in at
                                /*  the end of pass 1 with the number of
                                /*  global declarations.
        0       0       032             alloceglobs
                        000

                0                       procedure main()
                                                dclelob z

        02      1       006             dclloc  x
                        000
        04      2       006             dclloc  y
                        000

        06      2       106             dumpcv          /* dump proc stack
```

```
/-----------------------------------------------
/    An abbreviated listing of the console vector before
/    the call:
/
/      CONSOLE VECTOR:
/              07      pc
/              3       stkptr
/              071636  rtnptr
/              071656  rtntop
/
/          RETURN STACK:
/              071636  0       previous stack level
/              071640  0       return address
/              071642  071650  name pointer
/              071644  0       stack level b4 call
/              071646  0       grid pointer
/              071650  'main.g'        procedure name
/
/-----------------------------------------------

07      3       006'
                002
011     4       006'
                002
013     5       006'
                377
015     2       114             calli  sor(x,y,z)
                003
                000000
021     2       120             return          /* this return will
                                                /*  exit CIML


        3               procedure sor(a,b,c)
022     4       006             dclloc  x
                000

024     4       106             dumpcv          /* dump console vector
                                                /*  which includes
                                                /*  procedure stack

/-----------------------------------------------
/    An abbreviated look at the console vector:
/
/      CONSOLE VECTOR:
/              025     pc
/              7       stkptr
/              071660  rtnptr
/              071672  rtntop
/
/          RETURN STACK:
/              071660  071636  previous stack level
/              071662  0       return address
/              071664  071650  name pointer
/              071666  0       stack level b4 call
```



```
/              071670  0       grid pointer
/
/              071636  0       previous stack level
/              071640  021     return address
/              071642  071650  name pointer
/              071644  073270  stack level b4 call
/              071646  0       grid pointer
/              071650  'main.g'        procedure name
/
/-----------------------------------------------

025     5       004             load    a
                103
027     4       065             bgtp    11      /* goto 11 if .GT.
                000
031     4       120             return

032     5       004     11:     load    b
                102
034     5       010             move    c
                102
                        /* note that stack levels need not
                        /*  be equal at return statments
036     5       120             return
```

In the above example, the "stack level b4 call" is the contents of the stack pointer as it should be after the return. The large number is due to the fact that the user's stack is at the high end of core and the contents of the pointer is an absolute core address.

Assembler Processing: The assembler treats RETURN as it would any unconditional branch statement in that the stack pointer is assumed ambiguous after the RETURN. The stack pointer is made "not ambiguous" if one of three things follow the return statement:

1. A procedure definition
2. A label which has been referenced earlier in the program
3. The set stack pseudo op ". ="

8. Screen Coordinate Related Instructions:

poscurs	load position registers for console text input
ldjsn	load joy stick function button number
setgrd	partition the screen into numbered grid areas
tstgrd	calculated grid number of coordinates from current grid
tscanxy	scan current entry for text containing coordinates
ldcn	load character number of coordinates from tscanxy

POSCURS -- position input cursor on terminal
276

Description: the top two levels of the stack are ponped and loaded into the positioning registers for text input at the terminal (y coordinate assumed to be the top level). The positioning registers indicate where to place the next text line entered by the user at the terminal. The y register is decremented by 20 after each line entered so that if text is to be entered one line under the other, the positioning registers do not have to be reset.

If a control "u" is typed, the input line is re-positioned one line below the start of the current line.
sp := sp - 2

Examples:
The following program demonstrates the POSCURS instruction. The input registers are initially set to (0,780) which is the top left edge of the screen. The interpreter will place each successive input line below the last until 20 lines have been received.

```
0     1   006          dclloc  cnt 20  /* local variable cnt
          024
02    2   006          loadi   0       /* load x on stack
          000
04    3   007          loadi   780     /* load y on stack
          001414
07    1   276          poscurs         /* position where
                                        /*  input will go
010   2   100  loop:   wait    dcontx  /* wait for text input
          002
          000002
014   1   030          pop     1       /* pop device 6
          001
016   1   334          rcvcontx        /* receive text input
017   1   023          dbrz cnt, done  /* after 20 lines, go
          000
          000000
                                        /*  to done
023   1   054          br      loop    /* loop
          363

                done:
```

Assembler Processing: None

LDJSN -- load joy stick button number
262

Description: The button number from the previous joy stick hit is loaded onto the stack. The button numbers range from 1 to 12. (The user inputs joy stick coordinates by depressing one of the twelve buttons on the joy stick box.)
sp := sp + 1

Examples:

```
0     1     262          ldjsn   /* load joy stick button number
```

Assembler Processing: None

SETGRD -- load grid onto return stack
112

Description: the table of integers following the opcode is loaded onto the current level of the return stack. If there is a grid already loaded at this level, it is removed prior to loading the new one. The table following the opcode consists of two byte lists. The first list contains the x coordinates; the second contains the y coordinates. Each byte list consists of a byte which contains twice the number of coordinates in the list (the number of bytes in list) and two bytes for each coordinate.

The grid partitions the terminal screen into grid areas. The origin of the screen is in the lower right hand corner. Each x coordinate partitions the screen along the x axis; each y coordinate partitions the screen along the y axis. The intersections of each of these partitions form the grid areas. All area exterior to the bounded region has a grid number of zero. The internal grid areas are numbered from one, left to right, top to bottom. (The upper most left grid area is 1; the lower right area has the highest grid number). At least two ordinates are necessary along each axis to form a bounded region. However, if no coordinates are given for a particular ordinate, the entire axis is considered to be the bounded linear region. For example, the grid x() y(10,20) partitions the screen into 3 areas. The area 10<y<20 has the grid number one associated with it. The remaining areas, y<10 and 20<y, have a grid number of zero.

Grids can be defined at any time and as many times in a program as one wishes. Grids are inherited by sub-procedures from the calling procedure. Because the grid is stored in the procedure stack, sub procedures can define new grids without affecting the grid which is current to the calling procedure. Hence

a grid stays current until the next SETGRD instruction or until a RETURN to a procedure with a different current grid.
sp := sp

Examples:

```
0     0     112          setgrd  x(10,20,30), y(100,200)
            006
            000012
            000024
            000036
            004
            000144
            000310
                  /* The assembler will accept many versions
                  /*  of the above.  A few of these are:
015   0     112          setgrd  (10,20,30) (100,200)
            006
            000012
            000024
            000036
            004
            000144
            000310
032   0     112          setgrd  (10 20 30) (100 200)
            006
            000012
            000024
            000036
            004
            000144
            000310
                  /* The assembler will also recognize the
                  /*  following forms:
047   0     112          setgrd  x y(100,200)
            000
            004
            000144
            000310
056   0     112          setgrd  y(100,200)
            000
            004
            000144
            000310
```

Assembler Processing: The assembler assumes that the first parenthesized list contains the x coordinates and the second, the y coordinates. However, if only y coordinates are to be given, the list should be identified with the "y" as in the last example.

TSTGRD -- compute and load grid number of the coordinates on the stack
266

Description: the x and y coordinates on the stack (y coordinate on top) are compared against the current grid. The computed grid number is loaded onto the stack. See SETGRD.
sp := sp + 1

Examples:
Suppose the following grid which defines 6 bounded areas on the screen is current.
x(10, 20, 30, 40) y(20, 40, 60)

The unbounded regions have a grid number of zero. The bounded regions are assigned consecutive numbers beginning with one in the upper left hand corner and proceeding left to right, top to bottom.
The point (15, 50) lies in the upper left most area or grid number one. The point (35, 50) lies in the upper right most area or grid number three. The point (50, 50) lies to the right of the maximum x coordinate and therefore has a grid number of zero.

Assembler Processing: None

TSCANXY -- scan current entry for text with coordinates that are on the stack
050

Description: the current entry is scanned for a text line which contains the coordinates that are on the stack (y coordinate on top). If a text line in the entry is found containing the designated point, the current text line number in the block header is set to the line, "cn" is set to the character number of the nearest character, and control passes to the address specified in the two bytes following the opcode. Each character cell is assumed to occupy an area 20 dots high by 12 long. The length of a text line is the number of characters in the line multiplied by 12. The upper left most point covered by the text in the entry has the coordinates:
x: the x coordinate of the text coordinates in the entry plus the value in "dx" in the block header.
y: the y coordinate of the text coordinates in the entry plus the value in "dy" in the block header plus 20.

For example, suppose that the current entry contains the following two text lines:
frontal lobotomy
took a lot out of me

Let the text coordinate in the entry be (10,200) and let "dx" and "dy" be zero for simplicity. Furthermore, suppose the top of the stack contains 191 and the second level contains 115. Then the statement:
TSCANXY FOUNDIT
will cause the current text line number in the block header to be set to 2, "cn" to be set to 2, and control to be transfered to statement with the label prefix "FOUNDIT". The character cell containing the coordinates is the first 'o' in took. The coverage stops at the last character in each text line and therefore if the coordinates represent a point before or after any text line in the entry, the scan will fail.
sp := sp

Examples:

```
0     1    007          loadi   115     /* x coordinate
           000163
03    2    007          loadi   191     /* y coordinate
           000277
                  /* Control will branch to "foundit" if the
                  /*  entry contains the point (115,191).
06    2    050          tscanxy foundit
           000000
011   3    004   foundit:  load .
           000

                  /* notice that at assembly time the address
                  /*  of "foundit" was not known.  Throughout
                  /*  the assembly listing, all forward address
                  /*  references appear as zero.
```

Assembler Processing: The symbol following the mnemonic is assumed to be a label indentifier. The absolute branch address is assembled in two bytes, the low order 8 bits are in the first byte. The example above demonstrates forward referencing. If a symbol is found and presumed to be a label which is not already in the symbol table, it is entered at this time. The current program address along other information is entered in the forward address list so that the branch address can be filled in at the conclusion of the first pass.

LDCN -- load character number on stack
264

Description: the internally stored "character number" is loaded onto the stack. The character number is set by the TSCANXY instruction and gives the character position in the current text line of the coordinates. See TSCANXY. (In the PDP-8 interpreter system "character number" also contains the cursor position in the last input line. The terminal user has the capability of moving the cursor about the line being typed in.)
sp := sp + 1

Examples:

```
0      1      264          ldcn     /* load "cn" on stack
```

Assembler Processing: None

9. IO Related Instructions:

wait	wait for input
setimer	arm timer interrupt
rcvtimer	receive timer input (interrupt)
rcvcontx	receive text line input from console
rcvconjs	receive joy stick input from console
sndcsotx	send text line to IBM360
sndcsofi	send file to IBM360
rcvcsotx	receive text line from IBM360
rcvcsofi	receive file from IBM360
rcvcsoip	receive interpreter program from IBM360
snd732tx	send text line to Interdata 7/32
snd732fi	send file to Interdata 7/32
rcv732tx	receive text line input from Interdata 7/32
rcv732fi	receive file input from Interdata 7/32
rcv732ip	receive interpreter program from Interdata 7/32

WAIT -- suspend execution and wait for input
100

Description: this instruction waits for input from selected devices. The first byte after the opcode contains the number of following bytes to be included in the WAIT instruction. The next two bytes are treated as a word and contain a bit for each of the 16 possible input devices such as joy stick, Interdata file, etc. If input is received from a device whose corresponding bit is set, then the device number is loaded onto the stack and control resumes at the instruction following the WAIT. If input is received from a device whose corresponding bit is not set, a RETURN instruction is simulated and execution begins at the instruction following the last CALL statement. In either case once the WAIT instruction recognizes that there is input from some device, no other input will be recognized until this input is explicitly accepted or flushed. Input is accepted using the appropriate RCVxxx instruction. See also FLUSH.

The WAIT instruction provides for additional handling of joy stick input in the form of specifying particular grid areas to wait on. The bytes included in the WAIT instruction after the two specifying the devices indicate which of the grid areas to wait on. The first of these bytes represents grid areas 1 to 8; the second byte represents grid areas 9-16; etc. If joy stick input is pending and grid areas have been specified, then the joy stick coordinates are checked against the current grid. If the bit corresponding to the grid area is set, the joy stick device number is loaded on the stack and execution resumes after the WAIT. If however the bit corresponding to the grid area is zero, a RETURN is simulated. Grid areas need not be specified, and if none are, the joy stick behaves as any other device.
sp := sp + 1

Examples:
The GAS assembler provides a mnemonic for each device. The bit corresponding to each device is as follows:

Device	Mnemonic	Bit	Device Description
1	dtimer	0001	timer input
2	dcontx	0002	text line input from terminal
3	dconjs	0004	joy stick input
4	dplltx	0010	text input from UNIX
5	dpllfi	0020	file input from UNIX
6	dpllip	0040	interpreter program from UNIX
7	d732tx	0100	text input from Interdata 7/32
9	d732fi	0200	file input from Interdata
10	d732ip	0400	interpreter program from 7/32
11	dcsotx	1000	text input from IBM 360
12	dcsofi	2000	file input from IBM 360
13	dcsoip	4000	interpreter program from IBM 360

< three devices are currently unassigned >

The following program example demonstrates the WAIT instruction format.

```
                    /* In the following assembly listing, the
                    /*  first byte after the opcode gives the
                    /*  remaining length of the instruction.
                    /*  The next two bytes have been listed as
                    /*  a word and contain the devices to wait
                    /*  on.  The grid area bytes comprise the
                    /*  remaining bytes.  The numbering for the
                    /*  grid areas goes from low order to high
                    /*  order (ie grid area 1 is the low order
                    /*  bit, area 8 is the high order bit).
0       1   100             wait    dtimer
            002
            000001
04      2   100             wait    dcontx, dconjs
            002
            000006
                    /* Notice that the least number of bytes
                    /*  needed to represent the grid areas is
                    /*  used.  The grid may in fact have 50
                    /*  or so areas but if the largest one
                    /*  specified is 16, only 2 bytes are
                    /*  necessary for the grid areas.
010     3   100             wait    dconjs(1,2,3,8,9,10,16)
            004
            000004
            207
            203
                    /* The following WAIT demonstrates the role of
                    /*  the comma in the instruction.  If a comma
                    /*  is the last token on the line besides any
                    /*  comment, the instruction is continued on
                    /*  the next line.  A comma can also be used
                    /*  as delimiters within the joy stick grid
                    /*  specifications.
                        wait    dtimer, dconjs(1,2 3,8, 10 12),
016     4   100                 dcontx
            004
            000007
            207
            012
```

For programming examples, see the language descripton.

Assembler Processing: The only identifiers that the assembler expects after the WAIT mnemonic are the device mnemonics listed above. The order is not important. When a device is listed, the corresponding bit is set in the device word. If the device is the joy stick, then the assembler will check for a left parenthesis. If one is given, then the grid areas are scanned with order again of no importance. A closing right parenthesis is expected.

SETIMER -- set timer
330

Description: the number of seconds on the stack is popped and used to set the timer. When the timeout occurs, the input is signalled in the WAIT instruction. See RCVTIMER.
sp := sp - 1

Examples:
The following program will get a timer interrupt every 20 seconds and display the message "got timer input" on the screen. If any other type of input is received, the message "extraneous input" is displayed below the timer message. Notice that the timer must be accepted (RCVTIMER) and that to interrupt on the timer again, one must do another SETIMER instruction.

```
                    /* wait for input and flush everything but
                    /*  the timer
0       1   006             loadi   20
                024
02      0   330             setimer         /* interrupt in 20 secs

03      1   100     init:   wait dtimer dcontx dconjs dplltx dpllfi
                002
                000037
07      0   076             testbr  1, timer /* is the input the js
                001
                000000
013     0   326             flush           /* no, so flush input
014     1   006             loadi   20      /* x coord
                024
016     2   006             loadi   100     /* y coord
                144
020     0   312             disptxt "extraneous input"
                020
                145
042     0   054             br      init    /* wait for joy stick
                337
            timer:
044     0   332             rcvtimer        /* clear input
045     1   006             loadi   20      /* x coord
                024
047     2   006             loadi   120     /* y coord
                170
051     0   312             disptxt "got timer intput"
                020
                147
073     1   006             loadi   20      /* wait another 20 secs
                024
075     0   330             setimer
076     0   054             br      init
                303
```


Assembler Processing: None

RCVTIMER -- receive timer input
322

Description: receives timer input. This instruction has the effect of telling the system that timeout has been acknowledged and of clearing the timer input so that no input is currently pending from the timer. That is to say, if a WAIT followed the RCVTIMER instruction, execution would be suspended until there was input from some other device. A user error results if an attempt is made to receive the timer input when there has not been a timeout. The device number returned by the WAIT instruction for the timer is 1. See SETIMER, WAIT.
sp := sp

Examples:
The following is an example of a possible infinite loop. If timer input is received prior to text input from the console, the WAIT returns the device number for the timer and control passes to "gotimer". At "gotimer" the timer input is not received and control branches back to the WAIT instruction. Input is pending (ie the timer) and control passes again to "gotimer" and the program has entered an infinite loop.

```
0       1   006             loadi   10      /* set timer for 10
                012
02      0   330             setimer         /*  seconds
03      1   100     loop:   wait    dtimer, dcontx
                002
                000003
07      0   076             testbr  1, gotimer
                001
                000000

013     0   334             rcvcontx
                    /*          .
                    /*          .
                    /*          .
                    /* If we fall thru the TESTBR instruction
                    /*  we have received console input before
                    /*  the timeout.  Otherwise, control enters
                    /*  an infinite loop and
                    /*  the text "got timer input" will
                    /*  be continuously displayed on the screen:

014     1   006     gotimer: loadi  10      /* display msg on screen
```

```
                012
016     2       006             loadi   100     /* y coord
                144
020     0       312             disptxt "got timer input"
                017
                147
041     0       054             br      loop
                340
```

Assembler Processing: None

RCVCONTX, RCVCONJS -- receive text line input, receive joy stick input from console
334, 336

Description: in either instruction, if input does not exist from the particular device, a user error will result.

RCVCONTX (334) the text line input from the user's console is loaded into the text buffer, "p1" is set to zero, "p2" and "ln" are set to the number of characters in the text line. The input condition is cleared which is to say that another text line must be entered by the user before responding to this type of input again. (The device number returned by the WAIT instruction for console text input is 2)
sp := sp

RCVCONJS (336) the joy stick coordinates are loaded onto the stack. The y coordinate is on the top level. (The device number returned by the WAIT instruction for joy stick input is 3)
sp := sp + 2

Examples:

```
                        /* wait for console input in the form of text
                        /*  or joystick input.  If there is any other
                        /*  input, the WAIT will simulate a "RETURN".
0       1       100     loop:   wait    dcontx, dconjs
                002
                000006
                                                /* device # on stack
04      0       074             jmptble loop, textinput, jsinput
                006
                000000
                000000
                000000

                        textinput:      /* received text input
```



```
014     0       334             rcvcontx
015     0       246             apptl           /* append text line in
                                                /*  current entry
016     0       310             disptl          /* display current text
                                                /*  line
017     0       054             br      loop
                357

                        jsinput:        /* received joy stick input
021     2       335             rcvconjs        /* get coords on stack
022     0       312             disptxt "o"     /* display an 'o' at coords
                001
                157
025     0       054             br      loop
                351
```

Assembler Processing: None

FLUSH — flush current active input
326

Description: Input causing the resumption of execution from a WAIT instruction is considered the current "active" input. Execution will continue through WAIT instructions until the input is either explicitly received with the proper RCVxxx instruction or the input is FLUSHed. See WAIT.
sp := sp

Examples:

```
                        /* wait for input and flush everything but
                        /*  the joystick
0       1       100     init:   wait dtimer dcontx dconjs dpltx dpltfi
                002
                000037
04      0       076             testbr  3, joystick /* is the input the
                003
                000000
                                                /* joy stick?
010     0       326             flush           /* no, so flush input
011     0       054             br      init    /* wait for joy stick
                365
                        joystick:
013     2       335             rcvconjs        /* accept joy stick coords
```

Assembler Processing: None

10. Debugging Instructions:

traceon	turn program trace facility on
traceoff	turn program trace facility off
dumpaf	dump current active file
dumpcv	dump console vector containing both stacks, pc, sp, etc.

TRACEON, TRACEOFF -- turn trace facility on, turn trace facility off
103, 102

Description: TRACEON (103) a debugging instruction which turns the program trace facility on. Until the trace is turned off, the program address or counter and the opcode of each executed instruction is entered in the interpreter log file "glog".
sp := sp

TRACEOFF (102) a debugging instruction which disables the program trace facility.
sp := sp

Examples:

```
0      1     006           dclloc  x  2    /* declare local variable
             002
02     1     103           traceon         /* turn trace on
03     2     006           loadi   10      /* load constant 10
             012

05     3     006'
             005
07     2     017    loop:  add     5       /* add 5 to top level
010    2     023           dbrz    x, done /* dec x and branch on
             001
             000000                        /*  zero to done
014    2     054           br      loop
             367

016    2     102    done:  traceoff
```

```
/-----------------------------------------------
/
/    The above trace produces the following output
/    in the log file:
/
/       pc = 03       opcode = 006
/       pc = 05       opcode = 006
/       pc = 07       opcode = 017
/       pc = 010      opcode = 023
/       pc = 014      opcode = 054
/       pc = 05       opcode = 006
/       pc = 07       opcode = 017
/       pc = 010      opcode = 023
/       pc = 016      opcode = 102
/
/-----------------------------------------------
```

Assembler Processing: None

DUMPAF -- debugging instruction which dumps the current active file into the interpreter log file
104

Description: this instruction dumps the current active file into the interpreter log, "glog", with an explanation of each word in the file. Active files are in the users sub directory "GIML/". The name associated with the active file is listed at the beginning along with the program address of the DUMPAF instruction.
sp := sp

Examples:
The following program constructs a file which is dumped using the DUMPAF instruction.

```
0       1     006          loadi 0
                000
02      0     202          initf
03      1     006          loadi   1       /* block number
                001
05      2     006          loadi   2       /* # of info words/entry
                002
07      3     006          loadi   1       /* # of xy quads/entry
                001
011     0     210          initb           /* create block 1
012     0     232          appe
013     0     034          tbappt  "TERMINAL "
                011
                124
                105
                122
                115
                111
                116
                101
                114
                040
026     0     246          apptl
027     1     006          loadi   040
                040
031     0     124          stw     1       /* store info word
                001
033     1     007          loadi   600     /* x2
                001130
036     2     007          loadi   500     /* y2
                000764
041     3     007          loadi   400     /* x1
                000620
044     4     007          loadi   300     /* y1 of quad 1
                000454
047     3     122          stq     1       /* store lst xy quad
                001
051     2     122          stq     2
                002
053     1     122          stq     3
                003
055     0     122          stq     4
                004
057     0     104          dumpaf
```


The above DUMPAF instruction produces the following output in the log file:

```
ACTIVE FILE 0  'GIML/ACTFILA'    pc = 057
file length=23

disp   contents
0      1      current block number
1      21       disp to end of block
2      1        block number
3      1        current entry number
4      1        current text line number
5      2        number of info words/entry
6      1        number of xy quads/entry
7      0        dy
8      0        dx
9      13         disp to end of entry
10     32         info word 1
11     0          info word 2
12     300        y1 of quad 1
13     400        x1
14     500        y2
15     600        x2
16     0          y coord of text
17     0          x coord of text
18     9 124      char cnt, 'T'
19     105 122            'LF'
20     115 111            'MI'
21     116 101            'NA'
22     114 040            'L '
```

Assembler Processing: simple one byte instruction.

DMPCV -- debugging instruction which dumps the console vector into the interpreter log file
106

Description: the user's console vector is dumped into the interpreter log with suitable descriptors.

```
sp := sp
```

Examples:

The following program loads data on the stack and appends text to the text buffer in the console vector. The expansion of the text string in the following example has been abbreviated with the "-s" option of the GAS assembler.

```
0       1       006             loadi   1
                001
02      2       006             loadi   2
                002
04      3       006             loadi   3
                003
06      3       034             tbappt  "TERMINAL "
                011
                124
021     3       106             dumpcv
```

The progam counter "pc" is incremented after each fetch. Hence in the dump of the console vector, the "pc" always points to the next instruction.

```
CONSOLE VECTOR:
        022     pc
        0       curafnum
        0       lastjs
        0       cn
        0       x coord of text cursor
        0       y coord of text cursor
        3       stkptr
        071636          rtnptr
        071656          rtntop
        0       p1
        9       p2
        9       ccnt

    TEXT BUFFER
TERMINAL

    STACK:
        -3      0
        -2      0
        -1      0
        0       03
        1       02
        2       01

    RETURN STACK:
        071636          0       previous stack level
        071640          0       return address
        071642          071650  name pointer
        071644          0       stack level b4 call
```



```
        071646          0       grid pointer
        071650          'gas.bin'       procedure name
```

Assembler Processing: None.

11. Assembler Directives

procedure	procedure declaration
dclloc	local variable declaration and initialization
<id> =	local variable equal
dclglob	global variable declaration
allocglobs	allocate space for global variables
. =	set stack level pseudo op

PROCEDURE -- pseudo op for procedure definition
none

Description: an assembler pseudo op which signals the end of the current procedure and the definition of a new one. All local variables, formal parameters and labels currently in the symbol table are removed. The indentifier following the PROCEDURE pseudo op is a new procedure identifier which is global to the entire program. The identifiers found between the enclosing parenthesis are formal parameters and have a scope local to the new procedure.
sp := number of formal parameters

Examples:

```
                                /* the following procedure has 3 formal param-
                                /*  eters: n, m, filnum.  These are local to
                                /*  the procedure.
        3                       procedure finite(n,m,filnum)
0       4       006             dclloc  x       /* local variable
                000
02      5       006             dclloc  y       /* local variable
                000
                                dclglob z       /* global variable

04      6       004     loop:   load    x
                001
06      6       016             add     y
                001
010     6       010             move    x
                002
012     5       063             bgep    loop    /* branch if >=
                370
014     5       120             return


        2                       procedure input(n, m)
015     3       004             load    n
                101
017     3       016     loop:   add     m
                101
021     3       066             ble     loop
                374
023     3       120             return


        0                       procedure display()
024     1       006             dclloc  x       /* local variable
                000
026     2       006             dclloc  y       /* local variable
                000

030     3       004             load    z       /* load global z
                000
```

```
032     3     010          move    x       /* store it into x
                002
034     3     120          return
```

Assembler Processing: All symbol table entries with the attribute LOCALSCOPE are removed from the symbol table. The assembler then scans for the procedure name and a symbol table entry is created for the identifier with attribute PROC_DEF if one does not exist. The stack pointer is initialized to zero. The assembler then scans for a left parenthesis. Each identifier found before the closing right parenthesis is given the attribute PARMSTKOP (formal parameter) and assigned the current value of the stack pointer which is then incremented. In this way the level associated with the first formal parameter is 0; the level associated with the second is 1, etc.

DCLLOC -- declare local variable
same opcode as LOADI

Description: this instruction is actually a combination of the GIML instruction LOADI and the assembler pseudo op (<id> = .) for local variable definition.
sp := sp + 1

Examples:

```
0       1     006          dclloc  x       /* create local var x
                000
02      2     006          dclloc  y  03   /* create loc var y
                003
                                           /*  and initialize with
                                           /*  octal 03.

04      3     006          dclloc  z -10   /* create loc var z
                366
                                           /*  and initialize
                                           /*  to -10.

06      4     006          dclloc  A_LOC_VAR  0  /* create loc var
                000
                                           /* "A_LOC_VAR"

010     5     007          dclloc  $xyz3   32000  /* notice
                076400
                                           /* the generated LOADI
```



```
                                           /*  is for a 16 bit #
```

Assembler Processing: The form of the instruction is an identifier followed by an optional number. If no number is given, a zero is loaded into the space created for the variable. The level associated with the variable is the value of the stack pointer after the LOADI (In the above example the stack address associated with $xyz3 is 5). Any alpha-numeric string beginning with an alpha, '$' or ' _' is a valid indentifier as long as the id is not a permanent symbol, local variable, procedure id or label.

<id> = <ls_expression>   -- local variable declaration
none

Description: pseudo op which declares <id> to be a local identifier and which addresses the stack level given by the local stack expression on the right hand side. The variable has a scope local to the current procedure. The pseudo op is similar to the DCLLOC instruction except that the DCLLOC creates a level on top of the stack and assigns the identifier to the new level. See DCLLOC.
sp := sp

Examples:

```
0       1     006          dclloc  x       /* declare x to
                000
                                           /*  be local

02      2     006          loadi   10      /* load constant 10.
                012
              y = .                        /* declare y to local
                                           /*  and refer to level
                                           /*  2.
04      3     004          load    y       /* load y on stack
                000
06      4     006          loadi   -2      /* load constant -2
                376
              z = y + 1                    /* equate loc var z
                                           /*  to be 1 level
                                           /*  above y.
              w = .+1                      /* declare loc var w
                                           /*  to be one level
                                           /*  above top of stack
010     5     004          load    z
                001
```

```
012    6     004          load    w       /* w is the current
             000
                                         /*  top level of the
                                         /*  stack

/-----------------------------------------------
/    stack appears as follows:
/
/      6       10  <-- stack pointer
/      5    w: 10
/      4       -2
/      3    z: 10
/      2    y: 10
/      1    x:  0
/
/-----------------------------------------------
```

Assembler Processing: The assembler scans the stack expression on the right and checks for exactly one local stack variable reference (ie any local variable or '.'). An entry is created in the symbol table for the identifier and given the attributes LOCALSTKOP and LOCALSCOPE. The stack level associated with the identifier is the value of the local stack level expression.

DCLGLOB -- declare global stack variable

none

Description: an assembler pseudo op which declares the identifier following the DCLGLOB to be a global type stack variable. Global variables can be referenced in any procedure after the declaration and if the assembler directive ALLOCGLOBS has been given at the beginning of the program, space will be created for the global variables at the base of the stack. The new global stack identifier is equated with the next free global stack location. The numbering for the global locations begins at 377 (-1) which is the base of the stack and proceeds to 200 (-128) giving 128 maximum global variables. Local variables begin on the stack immediately above the last declared global variable. See ALLOCGLOBS.

sp := sp

Examples:

```
0      0     032          alloccglobs     /* allocate space for
             000
                                         /*  global variables.
                         dclglob x       /* declare some global
```



```
                         dclglob y       /*  variables

       0           procedure main()
                         dclglob z       /* declare a global
02     1     006         dclloc  a   5   /* declare some locals
             005
04     2     006         dclloc  b       /*  Note that local vars
             000
06     3     006         dclloc  c  -1   /*  can be initialized
             377

010    4     004         load    z       /* get z onto stack
             375
012    3     011         movep   x       /* store into x
             377

014    3     120         return
```

Assembler Processing: The identifier following the DCLGLOB is entered in the symbol table and given the attributes GLOBALSTKOP and GLOBALSCOPE. The identifier is assigned the next free global location as indicated above.

ALLOCGLOBS(S) -- allocate space for globals

same as PUSH

Description: this may be considered an assembler pseudo op even though a machine instruction is generated. It directs the assembler to generate a PUSH instruction which will create space on the stack for all subsequently defined global symbols. The assembler counts all global definitions and at the conclusion of the first pass the number of globals is filled in after the generated push instruction. The maximum number of GLOBALS symbols is 128. An error message will be printed if ALLOCGLOBS does not precede all executable instructions.

This instruction is useful when external programs wish to reference global symbols in the main interpreted program. Since the stack address associated with a global variable is purely a function of how many global declarations precede it, one can define global variables in external procedures in the correct sequence to correspond with appropriate globals in the main procedure. The ALLOCGLOBS directive is not included in the external procedure because space has already been created on the stack. One should note that the number of globals declared in external procedures should be no greater than the number in the main procedure or else global variables (temp) at the end of the external procedure will coincide with local variables in the main procedure.

```
sp := sp
```

Examples:
In the following example the second byte of the program is filled in by the assembler at the end of pass one with the number of globals which in this case is 4:

```
0     0     032          allocglobs      /* allocate space for
            000
                                         /*  global characters.
                         dclglob x       /* declare some global
                         dclglob y       /*  variables
                         dclglob z

      0          procedure main()
02    1     006          dclloc  a 5     /* some locals--note that
            005
04    2     006          dclloc  b       /*  locals can be initial-
            000
                                         /*  ized.
                         dclglob num     /* and another global
06    2     120          return
```

Assembler Processing: The assembler checks for the program counter to be zero and generates a PUSH opcode followed by a null byte. A flag is set indicating that the second byte of the program is to be filled in with the number of global declarations.

". =" -- set stack level pseudo op
none

Description: assembler pseudo op which sets the stack level relative to its current value. The pseudo op is necessary in instances where a previously unreferenced label follows an unconditional program transfer such as a JMP, BR, or RETURN. The value of the stack after an unconditional transfer is meaningless and if no stack level has been associated with the label by means of a foreward reference, the assembler has no criteria on which to establish the value of the stack pointer. The ". = " pseudo op allows the programmer to resolve the stack level ambiguity by equating the new stack level to the value of some local variable or an expression involving a local variable. If the pseudo op is not used, the assembler will print a warning message and continue with the current stack level.

A local stack level expression is any arithmetic expression

involving constants, exactly one local stack variable, and the binary arithmetic operators '+' and '-'. The following three examples are valid, local stack level expressions:

```
.
. + 4
x - 1 /* where x is a local variable
```

In general, a program can be written such that no stack ambiguities exist. This is accomplished by placing sections of code which are entered strictly by a branch or jump after at least one reference to the label at the beginning of the section. If the first label of a section of code has been referenced atleast once, then the stack level has been established for the section of code. Also see BR.
sp := value of local expression

Examples:

```
0     1     006          dclloc  x       /* define loc var x
            000
02    1     054    start: br      done
            000

                   /* Notice that the following section of code
                   /*  follows and unconditional control transfer
                   /*  (BR) and that the label "15" has no
                   /*  previous reference.  The assembler gives
                   /*  the following warning message:

warning--stack level is ambiguous
04    2     004    15:    load    x       /* get x onto stack
            000
06    2     056           beq     end     /* branch if equal
            000
010   1     055           brb     start   /* go to start
            370

                   /* The following section of code is also
                   /*  after an unconditional branch and the
                   /*  label has no previous reference.  In
                   /*  this case, the stack has been set with
                   /*  the ". = " pseudo op:

      2            . = . + 1               /* ( or . = x + 2)
012   1     030    16:    pop     1       /* pop one level
            001
014   1     072           jmp     start
            000002

017   1     030    end:   pop     1
            001
021   2     006    done:  loadi   10
            012
```

Assembler Processing: The assembler scans the expression on the right and tests for exactly one local variable having been specified ('.' is a local variable). The stack pointer is adjusted accordingly. If an attempt is made to adjust the stack when the stack pointer is not ambiguous, say after a LOAD or BEQ, an error will result.

Instruction Opcodes and Assembler Formats

Appendix A

The following is a summary of the GTIL instruction set. Opcodes are listed in octal. The initialization file for the GAS assembler can be formed from this table by removing lines containing the second occurance of a mnemonic and deleting the last two columns and headings.

Key to symbols:

s	stack level expression
#	integer (can be signed)
l	label
p	procedure name
a	formal parameter identifier
dev	device mnemonic
"text"	any text string
tl	length of text (# of characters + 1)

opc	mnemon	stack change	parse class	instr length	assembler format	
004	load	+1	25	2	load	s
006	loadi	+1	0	2	loadi	#
007	loadi	+1	0	3	loadi	#
010	move	0	2	2	move	s
011	movep	-1	2	2	movep	s
012	or	0	2	2	or	s
013	or	-1	2	1	or	
014	and	0	2	2	and	s
015	and	-1	2	1	and	
016	add	0	2	2	add	s
017	add	-1	2	1	add	
020	cmp	+1	2	2	cmp	s
021	cmp	0	2	1	cmp	
022	ibrz	0	4	4	ibrz	s, l
023	dbrz	0	4	4	dbrz	s, l
024	ashift	0	3	2	ashift	#
025	ashift	-1	3	1	ashift	
026	cshift	0	3	2	cshift	#
027	cshift	-1	3	1	cshift	
030	pop	-#	9	2	pop	#
032	push	#	9	2	push	#
034	tbappt	0	5	1+tl	tbappt	"text"
035	tbappt	0	5	1	tbappt	
036	tbinst	0	5	1+tl	tbinst	"text"
037	tbinst	0	5	1	tbinst	
040	tbcmp	+1	5	1+tl	tbcmp	"text"
041	tbcmp	+1	5	1	tbcmp	
042	tbmpl	0	3	2	tbmpl	#
043	tbmpl	-1	3	1	tbmpl	
044	tbmp2	0	3	2	tbmp2	#
045	tbmp2	-1	3	1	tbmp2	

opc	mnemon	stack change	parse class	instr length	assembler format	
046	tsearch	0	12	3+tl	tsearch	"text", 1
047	tsearch	0	12	3	tsearch	1
050	tscanxy	0	24	3	tscanxy	1
054	br	0	20	2	br	l
055	brp	-1	20	2	brp	l
056	beq	0	7	2	beq	l
057	beqp	-1	7	2	beq	l
o60	bne	0	7	2	bne	l
061	bnep	-1	7	2	bnep	l
062	bge	0	7	2	bge	l
063	bgep	-1	7	2	bgep	l
064	bgt	0	7	2	bgt	l
065	bgtp	-1	7	2	bgtp	l
066	ble	0	7	2	ble	l
067	blep	-1	7	2	blep	l
070	blt	0	7	2	blt	l
071	bltp	-1	7	2	bltp	l
072	jmp	0	6	3	jmp	l
074	jmptble	-1	17	2+2*n	jmptble	l1,l2,...,ln
076	testbr	-1	11	4	testbr	#, l
100	wait	+1	10	4+n/8	wait	dev1,dev2,...,js(1,4,..,n)
102	traceoff	0	1	1	traceoff	
103	traceon	0	1	1	traceon	
104	dumpaf	0	1	1	dumpaf	
106	dumpcv	0	1	1	dumpcv	
112	setgrd	0	13	3+2*(m+n)	setgrd	x(x1,x2,..xn) y(y1,...ym)
114	calli	-n	14	4	calli	p(a1,a2,...,an)
116	calle	-n	15	4+tl	calle	"text"(a1,a2,...an), 1
117	calle	-n	15	4	calle	(a1,a2,...an), 1
120	return	0	21	1	return	
122	stq	-1	8	2	stq	#
123	stq	-2	8	1	stq	
124	stw	-1	8	2	stw	#
125	stw	-2	8	1	stw	
126	ldq	+1	8	2	ldq	#
127	ldq	0	8	1	ldq	
130	ldw	+1	8	2	ldw	#
131	ldw	0	8	1	ldw	
134	neg	0	1	1	neg	
136	com	0	1	1	com	
140	tbinit	0	1	1	tbinit	
142	tbpshc	+1	1	1	tbpshc	
144	tbinsc	-1	1	1	tbinsc	
146	tbdelc	0	1	1	tbdelc	
150	tbchngc	-1	1	1	tbchngc	
152	tbappc	+1	1	1	tbappc	

opc	mnemon	stack change	parse class	instr length	assembler format
154	tbdelt	0	1	1	tbdelt
156	tbcnvo	-1	1	1	tbcnvo
160	tbcnvd	-1	1	1	tbcnvd
162	tbldp1	+1	1	1	tbldp1
164	tbldp2	+1	1	1	tbldp2
166	tbldln	+1	1	1	tbldln
170	tbstp1	-1	1	1	tbstp1
172	tbstp2	-1	1	1	tbstp2
176	setfn	-1	1	1	setfn
200	ldfn	+1	1	1	ldfn
202	initf	-1	1	1	initf
204	setbn	-1	1	1	setbn
206	ldbn	+1	1	1	ldbn
210	initb	-3	1	1	initb
212	delb	0	1	1	delb
214	lddy	+1	1	1	lddy
216	lddx	+1	1	1	lddx
220	setdy	-1	1	1	setdy
222	setdx	-1	1	1	setdx
224	seten	-1	1	1	seten
226	lden	+1	1	1	lden
230	ldlsten	+1	1	1	ldlsen
232	appe	0	1	1	appe
234	inse	0	1	1	inse
236	dele	0	1	1	dele
240	settln	-1	1	1	settln
242	ldtln	+1	1	1	ldtln
244	ldlsttn	+1	1	1	ldlsttn
246	apptl	0	1	1	apptl
250	instl	0	1	1	instl
252	dell	0	1	1	dell
254	chngl	0	1	1	chngl
256	ldxyt	+2	1	1	ldxyt
260	setxyt	-2	1	1	setxyt
262	ldjsn	+1	1	1	ldjsn
264	ldcn	+1	1	1	ldcn
266	tstgrd	+1	1	1	tstgrd
276	poscurs	-2	1	1	poscurs
300	dispf	0	1	1	dispf
302	dispb	0	1	1	dispb
304	dispe	0	1	1	dispe
306	dispq	-1	1	1	dispq
310	disptl	0	1	1	disptl
312	disptxt	-2	5	1+tl	disptxt "text"
313	disptxt	-2	5	1	disptxt
314	erase	0	1	1	erase
320	fload	0	23	3	fload 1
322	fsave	0	1	1	fsave

opc	mnemon	stack change	parse class	instr length	assembler format
324	fdel	0	1	1	fdel
326	flush	0	1	1	flush
330	setimer	-1	1	1	setimer
332	rcvtimer	0	1	1	rcvtimer
334	rcvcontx	0	1	1	rcvcontx
336	rcvconjs	+2	1	1	rcvconjs
340	sndcsotx	0	1	1	sndcsotx
342	sndcsofi	0	1	1	sndcsofi
344	rcvcsotx	0	1	1	rcvcsotx
346	rcvcsofi	0	1	1	rcvcsofi
350	rcvcsoip	0	1	1	rcvcsoip
352	snd732tx	0	1	1	snd732tx
354	snd732fi	0	1	1	snd732fi
356	rcv732tx	0	1	1	rcv732tx
360	rcv732fi	0	1	1	rcv732fi
362	rcv732ip	0	1	1	rcv732ip
364	sndplltx	0	1	1	sndplltx
366	sndpllfi	0	1	1	sndpllfi
370	rcvplltx	0	1	1	rcvplltx
372	rcvpllfi	0	1	1	rcvpllfi
374	rcvpllip	0	1	1	rcvpllip

Form AEC–427
(6/68)
AECM 3201

U. S. ATOMIC ENERGY COMMISSION
UNIVERSITY–TYPE CONTRACTOR'S RECOMMENDATION FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

1. AEC REPORT NO.: COO-2383-0038

2. TITLE: GIML REFERENCE MANUAL

3. TYPE OF DOCUMENT (Check one):

- [x] a. Scientific and technical report
- [ ] b. Conference paper not to be published in a journal:
  - Title of conference
  - Date of conference
  - Exact location of conference
  - Sponsoring organization
- [ ] c. Other (Specify)

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

- [x] a. AEC's normal announcement and distribution procedures may be followed.
- [ ] b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.
- [ ] c. Make no announcement or distrubution.

5. REASON FOR RECOMMENDED RESTRICTIONS.

6. SUBMITTED BY: NAME AND POSITION (Please print or type)

C. W. Gear
Professor and Principal Investigator

Organization
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Signature: Charles W. Gear. Date: April 1977

FOR AEC USE ONLY

7. AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION.

8. PATENT CLEARANCE.

- [ ] a. AEC patent clearance has been granted by responsible AEC patent group.
- [ ] b. Report has been sent to responsible AEC patent group for clearance.
- [ ] c. Patent clearance not required.

BIBLIOGRAPHIC DATA SHEET

1. Report No.: UIUCDCS-R-77-857
2.
3. Recipient's Accession No.:

4. Title and Subtitle: GIML REFERENCE MANUAL
5. Report Date: April 1977
6.

7. Author(s): Neal M. Hennegan
8. Performing Organization Rept. No.:

9. Performing Organization Name and Address:
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
10. Project/Task/Work Unit No.:
11. Contract/Grant No.: US ERDA/EY-S-76-02-2383

12. Sponsoring Organization Name and Address:
Energy Research and Development Administration
9800 South Cass Avenue
Argonne, IL 60439
13. Type of Report & Period Covered: research
14.

15. Supplementary Notes:

16. Abstracts:

This manual describes in detail the instruction set and assembler directives of the graphical interactive machine language GIML. The instructions are grouped according to dependence on architectural features. Included in the discussion of each instruction are the mnemonic, octal code, description, examples, and notes on assemble processing. For further discussion of the machine architecture and language see GIMLI AND GIML--A MACHINE AND PROGRAMMING LANGUAGE FOR LOW COST INTERACTIVE GRAPHICAL SYSTEMS, Report No. Department of Computer Science, University of Illinois, Urbana, IL 61801.

17. Key Words and Document Analysis. 17a. Descriptors:

interactive machine language
graphics
programming language
scoping of input response
interpreter

17b. Identifiers/Open-Ended Terms:

17c. COSATI Field/Group:

18. Availability Statement: unlimited
19. Security Class (This Report): UNCLASSIFIED
20. Security Class (This Page): UNCLASSIFIED
21. No. of Pages: 124
22. Price:

FORM NTIS-35 (10-70)
USCOMM-DC 40329-P71

MAY 17 1977

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.852-857(1977
Extension of the Huffman tree construct
3 0112 088403388